Modeling Languages: metrics and assessing tools

Daniela Fonte, Ismael Vilas Boas, José Azevedo, José João Peixoto, Pedro Faria, Pedro Silva, Tiago Sá, Ulisses Costa, Daniela da Cruz, and Pedro Rangel Henriques

Department of Informatics, University of Minho
Campus de Gualtar, 4710-057 Braga, Portugal
{danielamoraisfonte, ismael.vb, jazevedo, jj.peixotopereira, pedro.faria.80, zepedro.cs, ulissesmonhecosta, danieladacruz, pedrorangelhenriques}@gmail.com, tiago@esterisco.com

Abstract. Any traditional engineering field has metrics to rigorously assess the quality of their products. Engineers know that the output must satisfy the requirements, must comply with the production and market rules, and must be competitive.
Professionals in the new field of software engineering started a few years ago to define metrics to appraise their product: individual programs and software systems. This concern motivates the need to assess not only the outcome but also the process and tools employed in its development. In this context, assessing the quality of programming languages is a legitimate objective; in a similar way, it makes sense to be concerned with models and modeling approaches, as more and more people start the software development process by a modeling phase.
In this paper we introduce and motivate the assessment of models quality in the Software Development cycle. After the general discussion of this topic, we focus the attention on the most popular modeling language – the UML – presenting metrics. Through a Case-Study, we present and explore two tools. To conclude we identify what is still lacking in the tools side.

Keywords: Modeling Languages, Software/Language Quality, Software/Language Metrics, UML

1 Introduction

Models are a representation of reality aiming at the simplification of some complex objects: they are built so that we can better understand the system being developed. They allow us to specify the structure and behavior of a system, providing the guidance lines/blueprints for constructing a system, and finally, they document the decisions taken for a given system. Specifying means building models that are precise, unambiguous and complete.

Some models are best described textually, other graphically. All interesting systems exhibit structures that transcend what can be represented in a programming language.

A modeling language is a language whose vocabulary and rules focus on the conceptual and physical representation of a system.

One the one hand, one can produce a strict formal specification of the system, which allows us to reason over the system proprieties, without running the system. On the other hand, one can follow a pragmatic approach, using a diagrammatic specification of the system, not allowing us to reason over programs, but deriving programs from the model specification. That aside, when assessing a modeling language we might need to infer on its quality.

> Effective management of any process requires quantification, measurement, and modeling. Software metrics provide a quantitative basis for the development and validation of models of the software development process. Metrics can be used to improve software productivity and quality [Mil98].

The main goal of using model/software metrics is to be able to generate quantifiable measurements from the specifications/software. The use of model metrics is even more important to numerous valuable applications in earlier stages of the development process: in scheduling, cost estimation, quality assurance, and personnel task assignments. Nowadays, this metrics become increasingly essential for Software Engineering: they are crucial even for reengineering processes. In *Forward Engineering* they are used to measure the software quality and estimate cost and effort of software projects [FP98]. In the field of *Software Evolution*, they can be both used to identify stable or unstable parts of a system as to determine where refactoring can be or have been applied [DDN00]. They even can be used for assessing the quality and complexity of software systems in *Software Reengineering* or *Reverse Engineering* [CC90].

When focusing on the field of Object-Oriented (OO) systems, many metrics have been proposed for assessing the design of a software system. However, most of the existing approaches involve the analysis of the source code and cannot be applied in earlier stages of the development process. In fact, it is not always simple to apply the existing metrics in this earlier stages. As the Unified Modeling Language, proposed by Booch, Jacobson and Rumbaugh [BRJ05] has became a standard for expressing, design and specify OO systems, applying metrics to these models enables an early estimate of development efforts, implementation time, complexity and cost of the system under development.

In this paper, we will introduce and discuss the major existing metrics for *UML* models, and focus on present a set of tools designed for measure *UML* projects. In what follows, Section 2 we describe the principal measurements applicable to the most popular *UML* diagrams. In Section 3 we present two of the best tools designed to extract metrics from *UML* models and the results of applying them a real case-study. Then, Section 4 is devoted to the metrics assessment process. We conclude in Section 5 with a comparison between the presented tools.

2 Applying Metrics To UML Models

An *UML* model can be made from different diagrams, each one with a distinct view of the system. We have, in one hand, Use Cases diagrams which expose the system functional requirements and how each user interacts with them. They are a good overview of what features the system offers to the end user. In the other hand, we use Class Diagrams for represent the blueprint of the application under the developer perspective: they illustrate which programming components a system has and how they related to each other. Package Diagrams describe how to group the classes and how these groups relate to each other (*package import*, *package merge*). Here we present some metrics related to this three fundamental diagrams and conclude the section by introducing some metrics for other not less important *UML* diagrams.

2.1 Object-Oriented Software: CK Metrics

One of the most popular suites of OO metrics was proposed by Chidamber and Kemerer [CK94] to capture different aspects of OO designs, including complexity, coupling and cohesion. As we can see in [MP06], they were posteriorly adapted for modeling languages and can be easly applied to *UML* class diagrams.

This suite is composed by six metrics: *Weighted Methods Per Class* (WMC), *Depth of Inheritance Tree* (DIT), *Number of Children* (NOC), *Response For a Class* (RFC), *Coupling between Object Classes* (CBO) and, finally, *Lack of Cohesion in Methods* (LCOM). We detail bellow each metric and its features.

Weighted methods per class (WMC) - This metric regards to the complexity of a class method, being equal to the sum of those methods complexities. There are two kinds of WMC metrics:

- $WMC1_1$ is computed from a class diagram by counting the number of methods in that class - considering each method as an unity;
- WMC_{cc} is computed by counting the number of methods in each class, based on the result of the McCabe Cyclomatic Complexity of each method.

Depth of inheritance tree (DIT) - This metric is equal to the maximum length from the class to the root of the inheritance, which could be defined as the depth of the class. It is computed by taking the union of all the class diagrams in a *UML* model and traversing the inheritance hierarchy of the class.

Number of children (NOC) - This metric represents the number of childs and descendants of a certain class. Can be obtained gathering all diagrams class, in a *UML* modulation, and checking all the inheritance relations of the class.

Response for a class (RFC) - This metric measures the number of methods that can be invoked by an object of a given class. It can be obtained from a class diagram and from behavior diagrams (e.g. sequence diagrams), which can inform of several methods of other classes that are invoked by each of the class methods.

Coupling between object classes (CBO) - Two classes are related if a method of a class uses a instance variable or method of another class. Thus, we can compute this metric by counting the number of classes to which the class is related and counting all kind of references of the attributes and parameters of the class methods. Though, it is possible to calculate a more precise value if behavioral diagrams are taken into account, since the usage of instance variable and invocation methods are additional information.

Lack of cohesion in methods (LCOM) - It measures the number of sets of instance variables accessed by every pair of methods of a given class, that has a non-empty intersection. For this, is essential to use the information of the usage instance variables by the methods of a class – i.e., since a class diagram does not have information about the usage, it is required a sequence diagram.

This set of metrics can be found and cited in several papers (like [MP06]) and represent the basis of all the existing metrics for OO systems.

2.2 Class Diagram and Package Metrics

These diagrams are used to describe the types of objects in a system and the relationships among them. They describe the structure of a system by showing the system classes, their attributes and methods or operations. Their quality have a huge impact on the final quality of the software under development, as they describe the general model of the system information.

Marchesi Metrics

Metric	**Description**
NC	Number of Classes
CL1	Weighted Number of Class responsibilities
CL2	Weighted Number of Class Dependencies
CL3	Depth of inheritance tree
CL4	Number of immediate subclasses of a given class
CL5	Number of distinct class

Table 1. Marchesi Class Diagram Metrics

Marchesi Metrics

Metric	**Description**
NP	Number of Packages.
PK1	Number of Classes
PK2	Weighted Number of responsibilities of a Class
PK3	Overall Coupling among Packages

Table 2. Marchesi Package Metrics

Measures like the *Number of Attributes in the Class*, the *Number of Operations in the Class*, *Number of Inherited Attributes*, *Number of descents/ancestors of a Class*, or even the *Number of Interfaces Implemented* are used both for indicate the system complexity and as an index of quality. Many works present several metrics for this diagrams [GPC00], [Eic06], [YWG04]. The OOA metrics

defined by Marchesi [Mar98] also contemplate Class Diagrams, as we can see in Table 1.

In UML, classes can be grouped into Packages to define subsystems or even for implementation purposes. The measurement of a package complexity is useful to forecast the development efforts of it. For that, we can measure properties like the *Number of Classes of a Package*, the *Total Number of Packages in the system*, or the *Number of Interfaces in the Package*. Marchesi [Mar98] suggests several Package Metrics as we can see in Table 2 that allow to estimate this complexity.

2.3 Use Case Metrics

Use Cases Diagrams are graphical representations of entities which interact with the system (*actors*) and operations that the system must perform for them. They define a sequence of actions which illustrate a specific way of using the system.

These diagrams are functional specifications, collected at the beginning of a system development process. They are crucial to an early estimate of the system complexity and its development efforts, as we can see by the UC metrics defined in several works like [KB02], [MAC05], [Rib01].

In fact, measuring the number of Use Cases, actors and communications among them is a good indicator of the system complexity, as well as to quantify the relationship between diagrams (i.e. estimates the number of UC that extend or include others).

One remarkable work on this area was performed by Michele Marchesi [Mar98]. Table 3 illustrates the Use Case metrics defined on this work.

Marchesi Metrics

Metric	**Description**
NA	Number of actors of the system.
UC1	Number of Use Cases in the system.
UC2	Number of communications among UC and Actors
UC3	Number of communications among UC and Actores without redundancies
UC4	Global complexity of the system

Table 3. Marchesi Use Case Metrics

The **UC4** metric represents a balance of the global complexity of the system, and its value is obtained through the values of **UC1**, **UC2** and **UC3** metrics.

2.4 Other Diagram Metrics

Statechart diagrams illustrate the behavior of an object. They define different states of an object during its lifetime, which are changed by events. A *state* expresses an action of an object during a certain time, when it does not receive external stimulus nor is there any change in its attributes.

Measures like the *Number of Entry Actions*, *Number of Exit Actions*, *Number of Transitions*, or even the *Number of Activities* are associated to the complexity and dimension of the problem [GMP02]. In Table 4 we can notice some examples from measurable attributes for this type of diagram.

Statechart Metrics

Metric	Description
TEffects	Number of transitions with an effect in the state machine.
TGuard	Number of transitions with a guard in the state machine.
TTrigger	Number of triggers of the state machine transitions.
States	Number of states in the state machine.

Table 4. Statechart Diagrams Example

Activity Metrics

Metric	Description
Actions	Number of activity actions.
Object-Nodes	Number of activity object nodes.
Pins	Number of pins on the activity nodes.
Guards	Number of guards defined on object and control flows of the activity.

Table 5. Example of Activity Diagrams

Activity diagrams describe work flows and are very useful for detail operations of a class (including behaviors expressed by parallel processing). As we can see in Table 5 several metrics for this diagrams are available.

Besides these metrics, it is possible to measure attributes like the *Number of Activity Groups/Zones*, the *Number of object flows* or even the *Number of Exceptions* of each diagram.

3 Tools for UML Metrics Calculation

Nowadays, it is very common to use tools like Visual Paradigm for UML[1] or even Poseidon for UML[2] for software application development. They offer a visual environment to model software, which reduces the complexity of software design. However, they do not support metrics specification - it is necessary to use other tools, designed for this task. In the next subsections, we will introduce the leading systems for quantitative analysis of the structural properties of UML models, and put them to test for exploring their features with a real case-study.

One of the tools that we are going to address is SDMetrics[3], a design measurement tool for UML models. Although its core is open source and available under the GNU Affero General Public License, SDMetrics GUI it is not freely distributed. It is a very complete design measurement tool, analyzing a wide range of UML diagrams, including Class, Use Case, Activity and Statemachine diagrams, generating several metrics for each type of diagram.

[1] Available at `http://www.visual-paradigm.com/product/vpuml/`
[2] Available at `http://www.gentleware.com/products.html`
[3] Available at `http://www.sdmetrics.com/`

The other tool we put to test is the Sparx System Enterprise Architect[4], a team-based modeling environment. It embraces the full product development lifecycle, supporting both software design, requirements management, and metrics calculation for Use Case Diagrams. Thus, it allow to estimate the complexity of the project in a earlier stage, as well as the complexity associated with each actor of the system.

Case Study

The case-study used to explore this tools is the modelation made for the *MWK (Manages With Knowledge)* project for the Software Systems Development subject during our graduation.

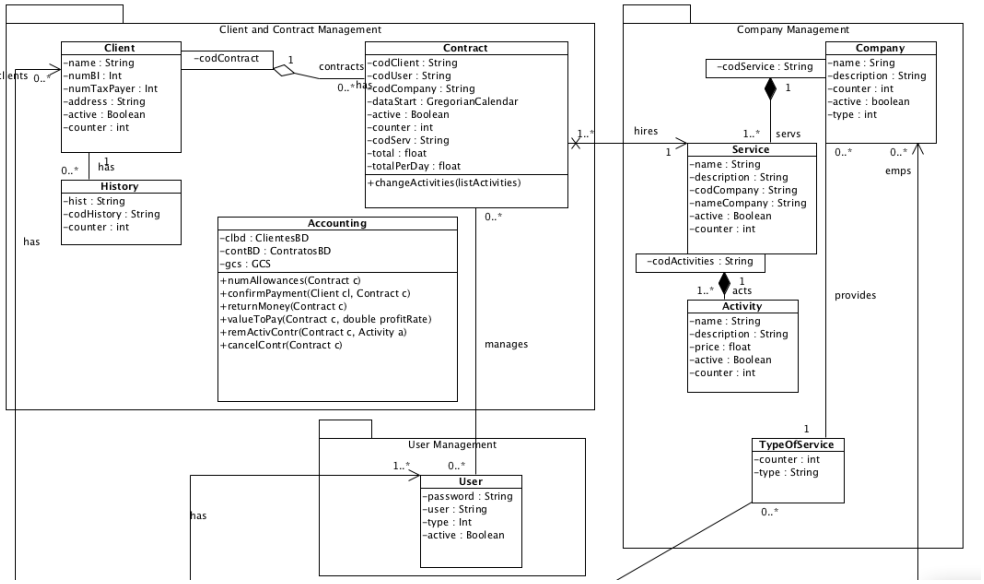

Fig. 1. Excerpt of a Class diagram

In this project, MWK is a company which does not provide services. In order to meet its clients needs it has a wide range of suppliers, that MWK subcontracts to being responsible for the service execution. Multiple suppliers can supply the same service and each service can be delivered in different ways. Each service can then be composed of multiple activities. As an example, there could be a service called *Shirts until 10 Kg* and inside this service there could be activities such as *wash, iron, sewing buttons, etc.* Each activity of a given service as a stipulated price, and can be hired by a client.

[4] Available at `http://www.sparxsystems.com.au`

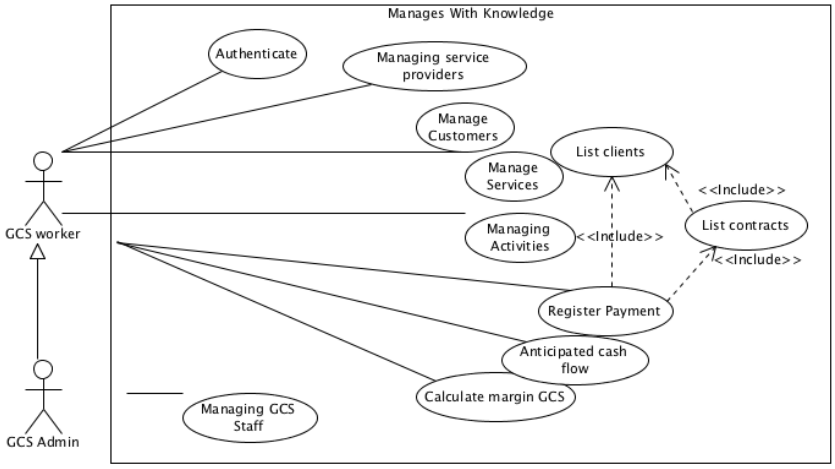

Fig. 2. The general Use Case model

As an example of the UML diagrams used to model this task, we can see in Figure 2 an image of a general Use Case diagram and, in Figure 1, an excerpt of a class diagram.

3.1 SDMetrics

SDMetrics is a design measurement tool for analyze a wide range of UML diagrams, including Class, Use Case, Activity and Statemachine diagrams.

For each type of diagram, this tool generates several metrics. For example, the **NumAttr** metric is calculated from Class diagrams and measures the number of attributes in a class. Other one is **ExtPts**, which is calculated from Use Case diagrams, and gives us the number of extension points of a given use case.

SDMetrics is written in Java, and provides us a graphical user interface for analyze *XMI*[5] files, most modeling tools support project exportation in XMI.

This tool allows to access the results from different views. We will introduce the ones that seem the most important:

- **Metric Data Tables** provides a table that matches each UML model element analyzed (table line) to its value for each metric (table column);
- **Histograms** provides a graphical distribution for each design metric;
- **Design Comparison** provides us a mean to compare the structural properties of two *UML* designs. It is very useful to compare the same design in different iterations of the development, or to compare an alternative design to the current one.

[5] *XMI* (**X**ML **M**etadata **I**nterchange) is an *OMG* (Object Management Group) standard to generate XML-based representations of UML and other OO data models.

– **Rule Checker** design rules and heuristics detect potential problems in the UML design such as incomplete design (i.e. unnamed classes, states without transitions, etc.); violation of naming conventions for classes, attributes, operations, packages; etc;
– **Catalog** this view provides us with the definitions of the metrics, design rules, and relation matrices for the current data set, and provides literature references and a glossary for them.

One of the most advanced features in this software is the possibility of defining Custom Design Metrics and Rules. The new metrics are defined in a XML file, with a very particular format, the *SDMetricsML* (SDMetrics Markup Language).

The SDMetrics tool does not provide a direct notion of good or bad quality of the design model. Despite that, on the SDMetrics User Manual[6] we can find several indications of how to interpret each kind of metrics.

Results Based on the *SDMetrics* manual, we will now explain how to interpret each kind of metric, analyzed by this software. Figure 3 illustrates some outputs of SDMetrics for our case-study. On the left, we can see an Histogram for class diagrams evaluating the *NumAttr* metric. On the right, we present an excerpt of a general metrics table for class diagrams.

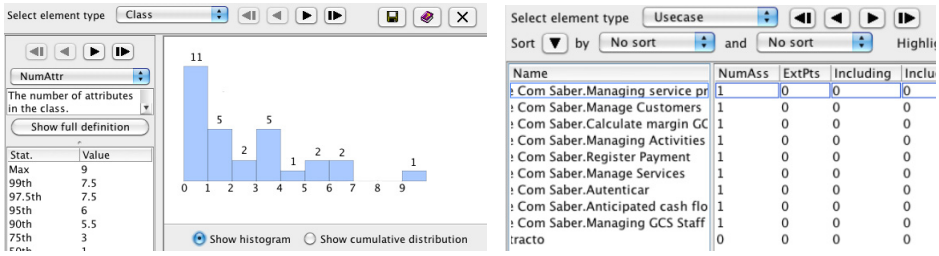

Fig. 3. SDMetrics: NumAttr Histogram and Metrics Table for Class Diagrams

Size metrics, which usually count elements inside design elements (class, package, etc), are good to estimate developing costs and effort, and can be directly found on the Metrics Table A large design element may indicate that it suffers from poor design, resulting in low functional cohesion. This has a negative impact on the understandability, reusability, and maintainability of the design element.

Coupling is the degree to which the elements in a design are connected: the more they are connected, the more they depend on each others. This high degree of dependency may affect the system maintainability, because when you change a design element, you may need to adapt the connected elements; and the system testability, because a problem in a design element may cause failure

[6] http://www.sdmetrics.com/manual/index.html

in a completely different connected element. Taking this in consideration, it is important to minimize coupling.

Inheritance-related metrics usually calculate features such as depth/width of the inheritance and number of ancestors/descendents of a design element. Such as high coupled elements, deep inheritance structures are believed to be more fault-prone. It is harder to fully understand a class that is situated deep in the inheritance tree, because you have to understand its ancestors. Also, modifying a design element with many descendants, may have a large impact on the system.

Complexity metrics measure the degree of connectivity between elements of a design element. They are concerned with relationships/dependencies between the elements in the design unit, such as the number of method invocations inside a class. The high complexity between the elements of a design element can make the design harder to understand, therefore more propitious to faults. Complexity metrics are usually strongly correlated with size measures. So even though they are good indicators of quality, such as fault-proneness, they provide no new knowledge comparing to size metrics.

3.2 Sparx Systems Enterprise Architect

Sparx Systems Enterprise Architect is another tool that provides modeling of UML diagrams. It supports mind map diagrams and project management, to provide full traceability from requirements specification to deployment end implementation. This tool also provides some metrics evaluation to compute the complexity of a project based on Use Case diagrams.

To perform this evaluation, the user needs to provide a level of implementation complexity for each interaction with authors. This task can be done when defining the Use Case descriptions or when performing the metrics evaluation.

To evaluate the metrics, Enterprise Architect has a wizard which enables other options for the complexity analysis. These options manipulate the *Technical* and the *Environment* complexities and are used to adapt the evaluation and perform a better result on estimation.

This system enables filtering the Use Cases used in evaluation, both on manual selection or regular expressions over use case name. This kind of filtering enables the project to be distributed and evaluated individually.

Results For use this tool in metrics calculation, the user have the possibility to do some tweaking over use case complexity (on their description) to get more precise results. This complexity admits simple values like low, medium or high.

Enterprise Architect offers a wizard which enables the edition of factors related to environment and technics complexity or even the hour rates, as we can see on the left side of Figure 4 Here we can use multiple factors to evaluate this factors, like usability or portability on technical complexity.

We can also access a Use Case metrics wizard, as we can see in the right side of Figure 4, which illustrates the set of default values used to evaluate the

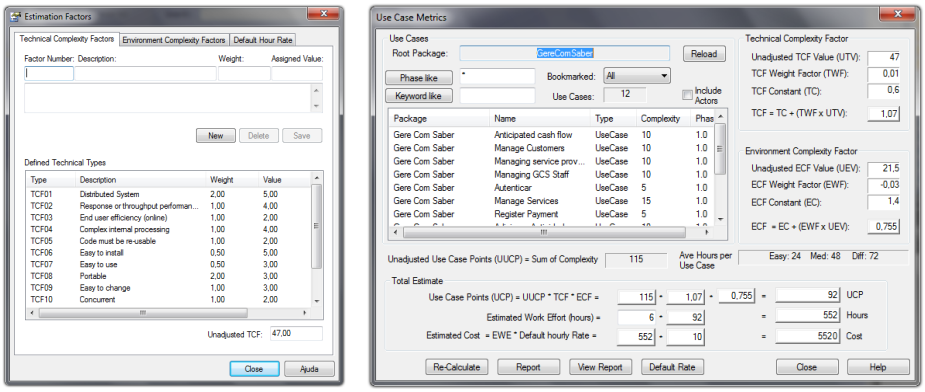

Fig. 4. Estimation Factors and Use Case Metrics of Enterprise Architect wizards

effort of the task. A set of predifined values are based on the factors edited on the previous wizard.

The final result of the metrics evaluation process is an estimation of Working Hours, Use Case Points [Rib01] and Total Cost needed to perform the system development.

In our case-study, the project has twelve Use Cases and many of them have medium complexity. Thus, as we can see in the right side of Figure 4, the effort to complete the task is 552 working hours, that would give a final result of €5.520. For obtaining this value only was changed the use cases complexity and everything else was left with default values.

4 Metrics Assessment

When interpreting the values obtained form measurements, one might question *is the metric really measuring the intended attribute?* this is a question that is pressent in the industry, yet unsuccessfully answered.

Working with models, one might want to know the quality of its model, i.e., which amount of the model really reflects object proprieties. To discuss model quality, one must use metrics to quantify those proprieties. Fenton [Fen99] estimates that companies spend about 4% of the development budget in the establishment of metrics programs, therefore, engineers should also certify and guarantee that the applied metrics actually quantify, measure, and model the attributes of the system.

Kaner and Bond try to propose a framework for metric evaluation, affirming that if a project or company is managed according to the results of inadequately validated measurement metrics, and those metrics are not tightly linked to the attributes they are intended to measure, distortions and disfunctionalities should be commonplace [KB04].

The industry, although not having a formal answer to this question, as advanced in this sense. The IEEE Standard 1061 [IEE98] defines an attribute as

"a measurable physical or abstract property of an entity". A quality factor is a type of attribute, *"a management-oriented attribute of software that contributes to its quality"*. A metric is defined as being a **measurement function**, and a **software quality metric** is defined as *"a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality"*. Any software metric must comply with the following criteria: correlation, consistency, tracking, predictability, discriminative power and reliability.

The IEEE Standard 1061 provides a sound layout of a methodology for developing metrics for software quality attributes.

5 Conclusion

Enterprise Architectis a full formal UML specification environment that also supports metrics. It is driven to enterprise market, thus oriented to minimize the cost and time spent with the production process, so the metrics evaluation offered is directed to this goal.

As mentioned, its metrics are oriented to available system functionalities and oriented to Use-case diagrams. We consider this a valid approach, but it requires some extra user interaction in the specification of the complexity of a task. This complexity could be archived by other means if other types of diagrams were also analyzed. The final results presented are simple and also oriented to the final cost, but if the complexity has been specified with knowledge of the environment, they could estimate very well the final cost of the implemented system.

Summing up, in metrics evaluation, this tool is a good choice to have an estimation of the implementation cost on a formally specified system, but this is its only goal, it cannot provide any kind of analysis about the quality of specification or do any automatic analysis about complexity of the system.

Acknowledgments

The authors would like to thank to the SDMetrics staff for provide us an Academic License to explore the full system features presented in this document.

References

Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The (2nd Edition)*. Addison-Wesley Professional, 2 edition, May 2005.

E.J. Chikofsky and II Cross, J.H. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13 –17, jan 1990.

S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20:476–493, June 1994.

Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *In Proceedings of OOPSLA '2000 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 166–177. ACM Press, 2000.

Holger Eichelberger. On class diagrams, crossings and metrics, 2006.

Norman E. Fenton. Software metrics: Successes, failures and new directions. 47(2-3), July 1999. pages 149-157.

N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, 1998.

Marcela Genero, David Miranda, and Mario Piattini. Empirical validation of metrics for uml statechart diagrams, 2002.

Marcela Genero, Mario Piattini, and Coral Calero. Early measures for uml class diagrams. *L'OBJET*, 6(4), 2000.

IEEE. Ieee standard for a software quality metrics methodology. December 1998.

Hyoseob Kim and Cornelia Boldyreff. Developing software metrics applicable to uml models. In *6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE*, 2002.

Cem Kaner and Walter P. Bond. Software engineering metrics: What do they measure and how do we know? (num.14-16), September 2004.

Parastoo Mohagheghi, Bente Anda, and Reidar Conradi. Effort estimation of use cases for incremental large-scale software development. In *In: Proc of the 27th Int'l Conf. on Software Engineering. St Louis*, pages 303–311, 2005.

M. Marchesi. Ooa metrics for the unified modeling language. In *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98)*, CSMR '98, pages 67–, Washington, DC, USA, 1998. IEEE Computer Society.

Everald E. Mills. Software metrics. December 1998.

Jacqueline A. Mcquillan and James F. Power. Some observations on the application of software metrics to uml models. Technical report, Department of Computer Science National University of Ireland, Maynooth, 2006.

Kirsten Ribu. Estimating object-oriented software projects with use cases. Technical report, University of Oslo, Dept, 2001.

Tong Yi, Fangjun Wu, and Chengzhi Gan. A comparison of metrics for uml class diagrams. *ACM SIGSOFT Software Engineering Notes*, 29:5, 2004.