24

# Applying Metrics for Quality Analysis and Improvement of Object-Oriented Software

**I. Morschel**
Daimler Benz Research Center
P.O. Box 23 60, D-89013 Ulm, Germany, +49 731 505 2870,
morschel@dbag.ulm.Dailmer-Benz.com

**Ch. Ebert**
Alcatel SEL AG
Lorenzstraße 10, D-70435 Stuttgart, Germany, +49 711 821 42283, cebert@stgl.sel.alcatel.de

**Abstract**

Software metrics are playing an important role in analyzing and improving quality of software work products during their development. Measuring the aspects of software complexity for object-oriented software strongly helps to improve the quality of such systems during their development, while especially focusing on reusability, reliability and maintainability. It is widely accepted that more widespread use of object-oriented techniques can only come about when there are tool systems that provide development support beyond visualizing code. Distinct complexity metrics have been developed and integrated in a *Smalltalk* development support system called *SmallMetric*. Thus, we achieved a basis for software analysis (metrics) and development support (critique) of *Smalltalk* systems. The main concepts of the environment including the underlying metrics are explained, its use and operation is discussed, and some results of the implementation and its application to several projects are given with examples.

**Keywords**

complexity metrics, development support, object-oriented metrics, quality control, Smalltalk.

## 1 INTRODUCTION

Software metrics are measures of development processes and the resulting work products. In this context we will focus on metrics that are applicable to software developed in *Smalltalk*. We will further concentrate on such metrics that can be used as quality indicators during the development process, hence providing support for the developers. These metrics are often classified as product metrics because their inputs are products of the development process. We will not distinguish metrics and measures from a mathematical point of view. When referring to

S. Bologna et al. (eds.), *Achieving Quality in Software*

complexity metrics we are using this phrase for a group of software metrics that measure structural or volume aspects of products that are intuitively related to parts difficult to understand. These difficulties in dealing with such complex components have been proved to cause high error-rates, testing effort and bad maintainability (for further details on metrics see Fenton (1991). Because of the extreme subjectivity of such quality attributes per se, it is important to select metrics that can be applied to the specific objectives of a project, that have been derived from the project's requirements and can be used to prove consistency, that can be applied during several phases of the development process on resulting products (design, code, documentation, etc.), and that can be collected and evaluated automatically by the development environment in use.

Product metrics are used to supply mechanisms for: Fenton (1991), Chidamber & Kemerer (1991), Pfleeger & Palmer (1990)

- estimating effort and costs of new projects;
- evaluating the productivity to introduce new technologies (together with their methods and tools);
- measuring and improving software quality;
- forecasting and reducing testing and maintenance effort.

Based on software metrics and quality data from finished projects quality models can be derived from all work products generated during the development process. Quality models, thus, are generated by the combination and statistical analysis of product metrics (e.g. complexity metrics) and product or process attributes (e.g. quality characteristics, effort, etc.). These models are evaluated by applying and comparing exactly those invariant figures they are intended to predict: the process metrics (e.g. effort, fault rate, number of changes since the project started, etc). Iterative repetition of this process can refine the quality models, hence allowing the use of them as predictors for similar environments and projects. While currently applied quality models for the software development process are primarily focussing on product metrics of procedural source code, developers are waiting for control mechanisms based on analysis metrics, hence being applicable much earlier. The obviously shorter feedback cycles permit a direct design improvement without waiting for the source code.

Due to successful application in many projects such metrics obviously should be available for object-oriented environments. The goals might be the same, primarily indicating potentially troublesome classes that should be improved before being introduced to the class libraries. The object-oriented paradigm could directly profit from metrics as a vehicle to instruct staff who are new to this approach. Furthermore software metrics could be used to measure the problems to introduce this paradigm and its acceptance as well as to set design standards for an organization.

Traditional metrics for procedural approaches are not adequate for evaluating object-oriented software, primarily because they are not designed to measure basic elements like classes, objects, polymorphism, and message-passing. Even when adjusted to syntactically analyze object-oriented software they can only capture a small part of such software and so can just provide weak quality indication, LaLonde (1994). It is hence important to define customized metrics for object-

oriented programs. Additionally the characteristics of the target language should be considered. Some languages directly support the object-oriented approach (*C++, Smalltalk, Eiffel*) and others just to some extent (*Ada*). Other factors like the size and contents of the class library and the semantics and syntactical form of particular commands should also be considered.

We will describe an programming analysis environment for *Smalltalk-80*, Goldberg & Robson (1983), because of its uniformity and elegance. The syntax of *Smalltalk* is easy to understand, it possesses a small number of operators (in contrast to *C++*), and it completely supports the notion of object, class, and inheritance. This article presents a basic set of metrics to support the development of object-oriented programs as well as a tool to automatically measure and to judge programs written in *Smalltalk*. In the next section the basic concepts of object-orientation and of *Smalltalk* will be presented. Section 3 gives an overview of related research in the area of object-oriented program analysis, quality control and product metrics. Section 4 and 5 describe the selection of metrics for object-oriented software and a tool environment called *SmallMetric*. Results from applying the analysis environment first to classroom projects and then in industrial projects are presented in section 6. A brief summary with an outlook on further work is given in section 7.

## 2 OBJECT-ORIENTATION AND *SMALLTALK*

Object-oriented modeling and programming is based on four fundamental concepts, namely inheritance, encapsulation, polymorphism, and reusability. for better understanding of the approaches described later, we'll try to give a rather brief summary about interesting features of object-orientation with respect to the *Smalltalk* programming language. As can easily be imagined the term "object" plays a central role in object-oriented programs. It comprehends data structures that describe its state, and methods[1] that realize its functionality. Data structures are encapsulated and provide information hiding with respect to their object which means that they do only offer access functions, called methods, but no direct use of the internal data structures. Objects communicate with each other via message passing which means that one method starts a method in another object. Mechanisms to hierarchically structure objects in classes exist in all object-oriented languages. Instances can be derived from classes and differ from other objects only on the basis of associated states.

Another important characteristic is the possibility of incrementally defining class hierarchies. This is done by the inheritance mechanism. From a superclass, a subclass inherits all its data structures and methods. In the subclass, new data structures and methods can be defined or they can be rewritten. *Smalltalk*'s inheritance mechanism for example is designed to model software evolution as well as to classify.

*Smalltalk* support the object-oriented concepts fully. It manipulates classes, objects and implements a single inheritance mechanism. It does not include multiple inheritance, prototypes, delegation, or concurrency in its standard version. In addition to its programming language *Smalltalk* includes an open programming

[1] we use in this paper the Smalltalk terminology.

environment to develop object-oriented programs. It offers a comfortable graphical user interface, several helpful tools and a vast class library. Programming language and environment coexist in a homogeneous form, where concepts at the programming level are reproduced in the environment. An example is the message passing mechanism. In *Smalltalk* programs, it means the activation of a method. The same strategy is used in the user interface to identify a selected object. This message passing consists of: an object (the receiver of the message), a message selector and optional arguments as parameters.

Object behavior is described by the mentioned methods, which have a selector and include *Smalltalk* commands. In different classes, methods with the same selector can exist. This is called polymorphism. The status of an object is captured through class and instance variables that might be accessed from outside by suitable methods. Class variables are defined at the class level and instance variables at the object level to store object's states.

## 3 ANALYSIS AND DEVELOPMENT SUPPORT FOR OBJECT-ORIENTED SOFTWARE

Most software developers apply quality assurance that identifies problems (as instances of poor quality) and ensures that they are resolved. Although striving to high quality standards, only a few organisations apply true quality control. Quality management includes continuously comparing observed quality with expected quality, hence minimizing the effort expended on correcting the sources of defect. In order to achieve software quality, it must be developed in an organized form including understanding of design and programming methods, associated review and testing strategies and supportive tools for analysis. The latter is accomplished through the use of metrics and statistical evaluation techniques that relate specific quantified product characteristics to some attributes of quality. The development environment provides the formal description of different work products that are analysed automatically. Multivariate analyses techniques provide feedback about relationships between components (e.g. factor analysis). Classification techniques help determining outliers (e.g. error-prone components). Finally, detailed diagrams and tables provide insight into the reasons why distinct components are potential outliers and how to improve them (see Ebert (1992) for a detailed description of such techniques).

So far there has been little work concerning the definition and use of metrics for object-oriented programs. With the broader application of this paradigm quality control, both analytic (i.e. with metrics) and constructive (i.e. by providing design and help facilities) quality control are of increasing importance.

One of the first attempts to investigate quality aspects of object-oriented programs was done by Lieberherr and colleagues, Lieberherr & Holland (1989). They defined a set of design rules that restricts the message-sending structure of methods. It was called *Law of Demeter*. Informally, the law says that each method can send messages to only a limited set of objects: to argument objects, to the *self* pseudovariable, and to the immediate subparts of *self* (self being the object or class itself). The *Law of Demeter* thus attempts to minimize the coupling between classes.

Most application of metrics for object-oriented programs are based on transforming well-known metrics for procedural programs, Karunanithi & Bieman (1993). Unfortunately such metrics do not cover completely the relevant aspects of coupling, such as inheritance or polymorphism. Other approaches suggest metrics that really focus on object-oriented descriptions, however do not offer any guidelines for using the metrics in practical projects, Bilow (1993), LaLonde & Pugh (1994). In the same context six metrics have been suggested for measuring elements contributing to the size and complexity of object-oriented design that will be used here as a base for selecting metrics, Chidamber & Kemerer (1991). An approach for estimating size and development effort of object-oriented systems based on requirements analysis has been discussed in Pfleeger & Palmer (1990). However, no metrics for distinct products have been provided. Sharble and Cohen (1993) compare two object-oriented development methods using an object-oriented brewery as example. They suggest indicators to enhance the software quality by increasing cohesion, reducing coupling, increasing polymorphism, and eliminating redundancy.

# 4 METRICS FOR ANALYSIS OF OBJECT-ORIENTED SOFTWARE

Goals such as quality improvement, increasing productivity or maturity certification are of growing interest in industry. Navigating the way with metrics is one important approach to ensure that a company stays on the course of achieving these goals. Though the search for underlying structures and rules in a set of observations is performed in software quality control and effective solutions to refine forecasting methods based on past data have been suggested, their applicability to object-oriented software development has been restricted.

Quality models are built on former project experiences and combine the quantification of aspects of software components with a framework of rules (e.g. limits for metrics, appropriate ranges etc.). For assessing overall quality or productivity, it is suitable to break it down into its component factors (e.g. maintainability), thus arriving at several aspects of software that can be analyzed quantitatively.

There is a growing awareness that such approaches could also support the object-oriented software development process. Anybody starting with object-oriented software rises the following questions, Lieberherr & Holland (1989), that also serve as guidelines for developing a measurement tool environment for quality control:

- What is good style in object-oriented programs?
- Are there any rules that can be applied to develop a good object-oriented program?
- Which metrics could be employed in order to determine if a program is "good" or not ?

Nevertheless, the mere definition of a metrics suite combined with statistical number crunching without intuitive backgrounds would result in the same acceptance problems procedural metrics applications ran into during the eighties, Fenton (1991). As long as the objectives of a object-oriented development process are not stated and supported with tailored methods, metrics would be of no practical

help. To overcome such problems we introduce *SmallMetric*, a tool to evaluate and meliorate object-oriented programs written in *Smalltalk*. It is embedded in an environment for the learning of object-oriented programming, Morschel (1993).

## 5 A DESCRIPTION OF THE OBJECT-ORIENTED METRICS FRAMEWORK

*SmallMetric* analyses object-oriented programs by applying construction rules that distinguish between (Figure 1):

- the static and dynamic structure of a class or an object;
- the static and dynamic relationships between classes and or objects.

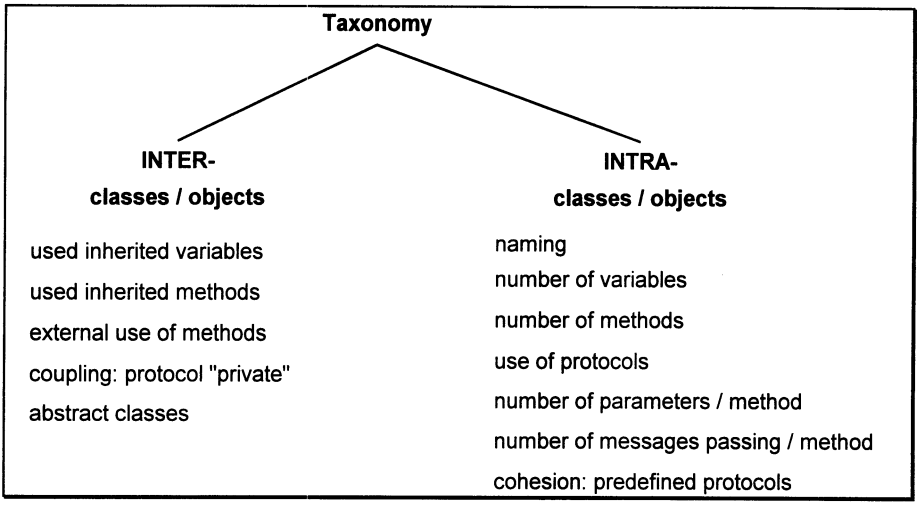


**Figure 1** Taxonomy for SmallMetric.

The metrics that are presented in the following list comprehend different aspects of object-oriented software. We will describe the underlying intuition of the metrics as well as a comprehensive summary of our observations from object-oriented development projects.

**Metric 1: Volume**

The volume of an object is a basic size measure that is intuitively connected with the amount of information inside a class. Many empirical investigations of metrics showed relationships among size metrics and comprehensibility or number of errors. Volume thus is a potential indicator of the effort to develop an object as well as for its future maintenance. The bigger the number of variables and methods, the more specific for one application. In other words, the object's reusability is likely to be small with increasing volume. Volume can be measured by:

- Number of variables (class and instance variables);
- Number of methods (class and instance methods).

**Metric 2: Method structure**

The internal structure of an object based on its methods and the information that is accessed by them is an indicator of its functionality. If the methods are overloaded with information to pass back and forth, there is good reason to assume that the object or class should be broken into several objects or classes. Method metrics are used to forecast effort for debugging and testing early. Method structure can be measured by:

- Number of parameters per method;
- Number of temporary variables per method;
- Number of message passing per method.

**Metric 3: Cohesion**

The term cohesion is frequently used in software engineering to designate a mechanism for *keeping related things together*. Cohesion can be defined to be the degree of similarity of methods. The higher the degree of similarity of methods in *one* class or object, the greater the cohesiveness of the methods and the higher the degree of encapsulation of the object. Cohesion in *Smalltalk* means the organization of methods, which set or access the value of a class or instance variable, under predefined schemes (*protocols*). These protocols are predetermined in *Smalltalk*. The programmer can use them to manipulate variables of an object. Such methods are called *accessors*, Beck (1993). The intuitive base is that direct reference to class and instance variables limits inheritance by fixing storage decisions in the superclass that can not be changed in a subclass. Besides, modifications in the structure of these variables are not visible to other methods, just to the accessors. Hence, the effort to extend or to modify a given program is minimized.

As an example consider an instance variable instVar of an object anObject. To access the class and instance variables it is necessary to define two kind of methods:

- one method for getting the value of an instance variable

```
instVar
    ^instVar
```

- and other for setting an instance variable

```
instVar: aValue
    instVar := aValue
```

This solution forces all accesses to variables to go through an accessor method. Therefore, *information hiding* with respect to variables and methods in a class is enforced, Parnas, Clements & Weiss (1985). *SmallMetric* examines a *Smalltalk* program to find accesses to variables outside of the predefined protocols. This is called a cohesion violation of an object.

**Metric 4: Coupling (Coupl)**

Coupling designates the interaction between objects that are not related through inheritance. Excessive coupling between objects besides inheritance is detrimental to modular design and prevents reuse. The more independent an object, the easier it is to reuse it in another project, Fenton (1991), Chidamber & Kemerer (1991). The suggested metric is:

- Number of invoked classes.

A predefined scheme in *Smalltalk* is the protocol **private**. It comprehends methods that should only be activated inside of an object. The *Smalltalk* compiler or interpreter does not check these specific accesses. When a message from another object starts a method under this protocol, undesirable effects can occur because during development such access had not been anticipated. *SmallMetric* tries to identify such references.

**Metric 5: Inheritance tree (Inh)**

This group of metrics analyzes the amount of inherited variables and methods used by a class. The use of inherited methods and data in a class indicates the difficulty of changing superior classes. On a low level of the inheritance tree variables and methods available to a class could be changed in meaning several times on higher levels, thus increasing complexity even more. It is hence necessary to provide information about how many methods and variables are available to a distinct class. The metrics are:

- inherited variables used;
- inherited methods used.

In *Smalltalk*, an instance variable can be directly set by an object of a subclass. This can reduce the reuse of a class in other applications. *SmallMetric* nominates it an "*information hiding violation*" (example 1).

**Metric 6: Class organization (Org)**

This group of analyses captures three comprehensability indicators: naming, checking of comments and the use of predefined protocols. Naming analyzes all identifiers of a class. *SmallMetric* informs the developer about their distribution. This metric has just documentation purposes. The existence of comments within an object is also checked. In *Smalltalk*, one can define a global comment to clarify the intents and functionality of an object. *SmallMetric* warns, when there is no such comment provided. The programmer may organize the methods of an object under predefined protocols. The *Smalltalk* environment advises the developer to use these recommendations which is checked by *SmallMetric*. For novices, these protocols can help to elucidate some aspects of a *Smalltalk* program.

```
Object subclass: #Superclass
    instanceVariableNames: 'text'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'SmallMetric'

Superclass subclass: #Subclass
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'SmallMetric'

Subclass class methodsFor: 'instance creation'
new

    ^super new initialize
```

```
Subclass methodsFor: 'initialize release'
initialize
    text := 'This is an example of an information hiding violation !!'
```

**Example 1** An information hiding violation

## 6 EXPERIENCES WITH *SMALLMETRIC* FOR *SMALLTALK* PROGRAM ANALYSIS

Upon starting SmallMetric a window is opened, which inquires the name of the class to be analysed. Wildcards (*) can be used. When the given class is found, a new window is created (Fig. 2). It presents the following information:

1. number and list of all variables;
2. number of methods;
3. buttons to switch between class and instance;
4. predefined protocols used;
5. naming;
6. violations of *SmallMetric* metrics.

Four buttons are provided to select a new class, to print the information of a class, to switch between different dialogue languages (now English and German) and to activate Help. The critique window of course can be adjusted to specific standards and process guidelines of an organization. It has a menu, which presents the design limits for development support.

*SmallMetric* comprises a basic set of guidelines for metric-based development support of *Smalltalk* applications. On the basis of the metrics above as applied to Smalltalk projects with available quality data, we extracted some *design guidelines* to enhance the quality of object-oriented programs written in *Smalltalk*. Because one of the main reasons for using object-oriented technology is reusability, we focussed our evaluations on maintainability and reusability. Such guidelines should be understood as recommendations and not as restriction of a programmer's creativity. The projects being analyzed ranged in size from few classes to 400 classes of a commercially available *Smalltalk*-based tool, thus covering effort of up to 30 person years. Our approach for extracting guidelines of metrics that can serve as indicators of poor quality is based on analyzing the classes with respect to complexity metrics and quality data. Since the metrics are applied on different scales (complexity metrics: ratio scale and quality metrics: ordinal scale) we performed non-parametric statistical methods for correlations and factor analysis. Unlike in other approaches we don't discard outliers, because it is - ex ante - unknown what classes or metrics are outliers. Instead all metrics are normalized with a quadratic approach before comparing or ranking them.

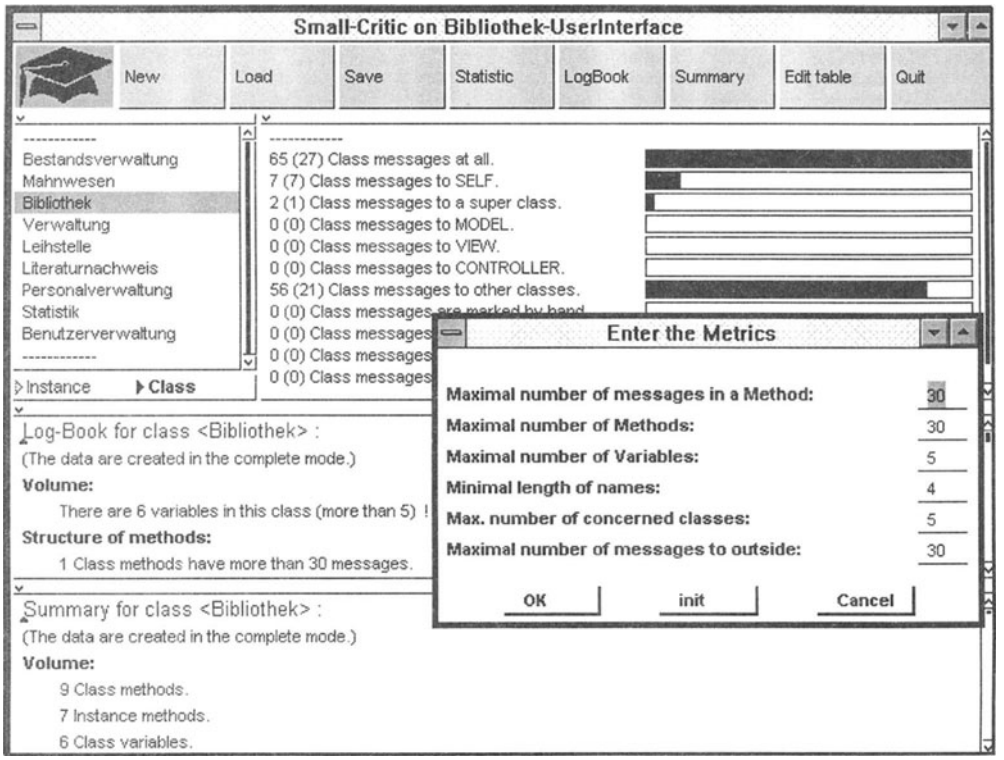


**Figure 2** The user interface of *SmallMetric*.

The following suggestions usually seem to be very clear in theory - but just have a look on your latest designs ...

- **Volume**:
  Number of variables → maximum 5
  Number of methods → maximum 30
  *Number of invoked classes and objects → is only numerically indicated*
- **Struct**:
  *Number of parameters per methods → is only numerically indicated*
  *Number of temporary variables per methods → is only numerically indicated*
  Number of message passing per methods → maximum 30
- **Cohes**:
  Existence of an *accessor* outside of a predefined protocol
- **Coupl**:
  *Number of external message passing per method → only numerically indicated*
  *External access of methods under the protocol private → only numerically indicated*
- **Inh**:
  Violation of the *information hiding* principle → *only numerically indicated*
- **Org**:
  Number of characters of an identifier → minimum 5
  Non-existence of comment

The measured values were analyzed with respect to boundaries (minimum, maximum), intervals, deviations from the average, and nonparametric correlations between them. The interpretation was performed according to these criteria and used as follows:

- Sustain a high comprehensability level by providing a sufficient length of descriptive parts in all design objects and object names with meanings, rather than enumeration such as "class-1". The descriptions should include subclasses or inheritance relations, changes of inherited methods or variables, functionality, related objects, used data items, date, author, test cases to be performed, requirements fulfilled, management activities and staff connected with this project.
- During class and object design, the metrics and their statistical evaluation (regarding similar projects) are taken to distinguish between different designs (e.g. alternative approaches, division into subclasses).
- During reviews at the end of design and coding, the metrics are taken as indicators for weak components (e.g. inadequate inheritance hierarchy, unsatisfying object description) and as indicators for process management (timely ordered number of classes or volume metrics).

After applying such metrics to different, however similar projects, the statistical results obtained can be used to define intervals or limits for metrics in order to increase quality.

# 7 SUMMARY AND FUTURE WORK

Most complexity metrics have been designed without regard to the problem domain and the programming environment. There are many aspects of complexity and a lot of design decisions influence the complexity of a product. This paper presents an approach to integrate software metrics with design support for object-oriented techniques based on *Smalltalk*. A tool environment for program analysis called *SmallMetric* that incorporates metrics and guidelines for improving programs has been developed. Based on these set of metrics we investigated different projects both from industry and academia to improve the guidelines. This approach to integrate a measurement tool system into *Smalltalk* illustrates a way to minimize the efforts for implementation and maintenance of such a tool and shows how to cope with changes in future requirements for such tools and their individual interfaces. By transforming the object-oriented information representation into another language it is possible to integrate such measurement techniques into other environments as well.

With an early analysis of software products we are able to provide developers with helpful hints to improve their designs and code during the development process and not at the end when it will be much more expensive. By following the given suggestions we could actually and reproducible improve designs and achieve better programs in terms of such quality items as understandability, reusability and maintainability. Of course, much more research is necessary in order to provide complete guidelines for achieving high quality designs. The basic step, however, still is the measurement and evaluation of software complexity as early as possible: during the software development process when the most expensive faults are induced (e.g. inheritance trees). By making software engineers aware that there are

suitable techniques and tools for analyzing their programs, even when they are object-oriented, this could be a small step to avoid a similar software crisis to what we are currently facing in procedural environments.

## ACKNOWLEDGEMENTS

The assistance of the Landis&Gyr corporation Switzerland, to provide product and process data of object-oriented projects is gratefully acknowledged. Several discussions with A. Riegg of Debis in Stuttgart contributed to the proposed guidelines.

## BIBLIOGRAPHY

Fenton, N.E.: *Software Metrics: A Rigorous Approach.* Chapman & Hall, London, UK, 1991.

Chidamber, S.R. and C.F. Kemerer: Towards a Metric Suite for Object Oriented Design. *Proc. of Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). Sigplan Notices*, Vol. 26, No. 11, Nov. 1991.

Pfleeger, S.L. and J.D. Palmer: Software Estimation for Object Oriented Systems. *Proc. Fall international Function Point Users Group Conference*, San Antonio, TX, USA, pp. 181 - 196, Oct. 1990.

Goldberg, A. and Robson, D.: *SMALLTALK-80 The Language and its Implementation.* Addison-Wesley, 1983.

Ebert, C.: Visualization Techniques for Analyzing and Evaluating Software Measures. *IEEE Transactions on Software Engineering*, Vol. 18, No. 11, pp. 1029 - 1034, Nov. 1992.

Lieberherr, K.J. and I.M. Holland: Assuring Good Style for Object-Oriented Programs. *IEEE Software*, Vol. 6, No. 9, pp. 38 - 48, 1989.

Bilow, S.: Software Entropy and the Need for Object-Oriented Metrics. *Journal of Object-Oriented Programming*, Vol. 5, Jan. 1993.

Sharble, R. and Cohen, S. The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods. *Software Eng. Notes,* Vol 18, No. 2, 1993.

Morschel, I.: An Intelligent Tutoring System for the Learning of Object-Oriented Programming. *Proc. EAEEIE '93.* Prague, 1993.

Beck,K.: To accessor or not to accessor? *The Smalltalk Report.* Vol. 2., Num. 8., June 1993

Parnas, D.L., P.C. Clements and D.M. Weiss: The Modular Structure of Complex Systems. *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, pp. 259 - 266, Mrc. 1985.

LaLonde, W. and J. Pugh: Gathering Metric Information Using Metalevel Facilities. *Journ. Object Oriented Programming*, Vol. 6, pp. 33 - 37, Mrc. 1994.

Karunanithi, S. and J.M. Bieman: Candidate Reuse Metrics for Object Oriented and Ada Software. *Proc. Int. Software Metrics Symposium.* IEEE Comp. Soc. Press, New York, pp. 120 - 128, 1993.

## BIOGRAPHY

Dr. I. Morschel: Majored in computer sciences at the UFRGS in Brazil (1983-1987), M.Sc. at the UFRGS (1988-1990), PhD under the supervision of the Department of Dialogue Systems at the University of Stuttgart (1990-1994) with thesis "Software Engineering - Object-Oriented Technology and Intelligent Tutoring Systems". Since 1994 employed at Daimler-Benz AG in the field of research of software engineering.