ELSEVIER

The Journal of Systems and Software 42 (1998) 89–102

The Journal of Systems and Software

Effort estimation and prediction of object-oriented systems [1]

P. Nesi [*], T. Querci

Department of Systems and Informatics, University of Florence, Via di S. Marta 3, 50139, Florence, Italy

Received 10 February 1997; received in revised form 24 March 1997; accepted 16 May 1997

Abstract

Due to the growing diffusion of the object-oriented paradigm (OOP) and the need of maintaining under control the process of software development, industries are looking for metrics capable of producing satisfactory effort estimations and predictions. These metrics have to produce results with a known confidence since the early phases of software life-cycle in order to establish a process of prediction and correction of costs. To this end, specific metrics are needed in order to maintain under control object-oriented system development. In this paper, new complexity and size metrics for effort evaluation and prediction are presented and compared with respect to the most important metrics proposed for the same purpose in the literature. The validation of the most important of these metrics is also reported. © 1998 Elsevier Science Inc. All rights reserved.

Keywords: Object-oriented metrics; Effort evaluation and prediction; Analysis and design metrics; Code metrics

1. Introduction

In recent years, several industries moved to the object-oriented paradigm (OOP) in the hope of satisfying their needs in terms of reusability, capability for programming "in the large" (i.e., for the capability of the OOP in modeling the reality by minimizing the cohesion among entities), etc., (Booch, 1994). In general, the introduction of the OOP is not immediate because it involves managers, analysts, designers, developers, etc. Thus, the adoption of the OOP implies to change the whole development process, (i.e., project management, resource evaluation, resource planning, requirements analysis, test, design, etc.). In most cases, the introduction of the object-oriented technology has not been accompanied by a corresponding effort to establish mechanisms for controlling the development process (Nesi, 1995). This lack is due on one hand to the lack of internal tradition in controlling the development process and, on the other hand, to the lack of consolidated suitable metrics for evaluating object-oriented projects.

In general, in order to guarantee the control of the development process, quantitative metrics for evaluating and predicting system characteristics must be used. One of the most important issues that should be maintained under control is the software cost (i.e., the effort). To this end, a linear/non-linear relationship between software complexity/size and effort (i.e., man-months or -days needed for system development, from requirements analysis to testing or in some cases only for coding) is commonly assumed. Therefore, the problem of effort evaluation is shifted to the problem of complexity or size evaluation. It should be noted that, when software complexity evaluation is performed after system building, it can be useful for: (i) predicting maintenance costs, (ii) comparing productivity and costs among different projects, (iii) learning the development process efficiency and parameters; when software complexity evaluation is performed before system building, it can be used for predicting costs of development, testing and early maintenance, etc.

Moreover, on the basis of the knowledge which is present in the early stages of the software life-cycle (e.g., number of classes, main relationships, number of methods, etc.) the process of code analysis allows the definition and tuning of metrics for predicting costs. From the cognitive point of view, the observable complexity can be regarded as the effort to understand subsystem/class behavior and functionalities. This complexity can be usually evaluated in the early phases and can be used for predicting costs of reuse and maintenance (Cant et al., 1991, 1994).

[*] Corresponding author. Tel.: +39-55-4796523; fax: +39-55-4796363; e-mail: nesi@ingfi1.ing.unifi.it; web: http://www.dsi.unifi.it/~nesi.

[1] This work was partially supported by MURST Ex60% govern Ministry of University and Scientific Research and by CESVIT, High-Tech Agency, Center for Software Quality, Florence.

Traditional *code metrics* for complexity/size estimation, often used for procedural languages (e.g., McCabe (1976), (Henderson-Sellers and Edwards (1990), Halstead (1977) and the number of lines of code (LOC)), are unsuitable to be directly applied for evaluating object-oriented systems (Henderson-Sellers and Edwards, 1990; Bilow et al., 1993; Nesi and Campanai, 1996). This is mainly due to the fact that working with the OOP leads to shift part of the human resources from the design/code phase to that of analysis (Henderson-Sellers et al., 1994). In that phase, classes are identified together with their main relationships [2] (*is_a*, *is_part_of*, *is_referred_by*). On the other hand, following evolutionary models for the development life-cycle (e.g., spiral (Boehm, 1986), fountain (Henderson-Sellers and Edwards, 1990), whirlpool (Williams, 1996), pinball (Ambler, 1994)), the distinction among phases is partially lost – e.g., some system parts can be under design when others are still under analysis. Therefore, these aspects must be captured with specific metrics, otherwise their related costs are unmeasurable (e.g., the costs of specialization, the costs of object reuse, etc.).

For eliminating the above mentioned drawbacks, some authors have defined new specific code metrics for evaluating size and/or complexity of object-oriented systems. Thomas and Jacobson have suggested to estimate class complexity as the sum of attribute and method complexities (Thomas and Jacobson, 1989), without considering the class external interface (i.e., method interface in terms of parameter complexity) and reuse. Henderson-Sellers has added to the above metric a term for addressing the problems of reuse (e.g., inheritance) (Henderson-Sellers, 1991). Chidamber and Kemerer have presented the Weighted Method per Class (WMC) which is the sum of the complexity of each class method and where each method complexity is obtained by using the McCabe complexity (Chidamber and Kemerer, 1994), by considering in this way only the functional and computational aspects of the class. Li and Henry have suggested to count the number of methods and the number of attributes (with the so-called *Size*2 metric) for addressing the problem of effort prediction (Li and Henry, 1993), without weights and, thus, tuning the model. In Henderson-Sellers (1994), Nesi and Campanai (1996) and Hopkins (1994) issues regarding the external and internal class complexity have been discussed by proposing several metrics. In Laranjeira (1990) and Lorenz and Kidd (1994) class complexity has been estimated by counting the number of LOC for all class methods. Most of the above mentioned metrics have been only partially validated.

[2] Following different methodologies, some of these relationships are defined only in the phase of design or also in that of analysis.

Recently, a growing attention on the process of software development has created the need to get process-oriented information and to integrate metrics into the software development process. Furthermore, owing to the presence of many differences among projects by the same company, it is also important to create an integrated environment for software development (editing, navigating among classes, measuring, etc.) and to perform project-oriented *tailored* measures. This means that it is important for a company to adopt a unique method and approach for project measurement, but this approach must be capable of being tuned in order to adapt its features to different types of projects and languages. This process of adaption is usually performed by adjusting weights and thresholds (Henderson-Sellers et al., 1994). Some studies with metrics and measurement frameworks for object-oriented systems have been presented in Laranjeira (1990), Meyer (1990), Henderson-Sellers (1993) and Coulange and Roan (1993), Li and Henry (1993), Brito e Abreu et al. (1995) and Nesi and Campanai (1996) where general concepts for the estimation of system size, complexity and reuse level have been proposed together with many other metrics.

In this paper, a research about the estimation and prediction of effort of object-oriented systems coded in C++ is presented. The metrics proposed belong to a framework specifically defined for C++ language. The main aspects of our metric framework have been inherited from that presented in Campanai and Nesi (1994) and Nesi and Campanai (1996) by the same research group. Moreover, due to the high number of metrics which are available in our framework (see Section 4), only those metrics which are related to software complexity, size and effort are reported and compared in this paper with respect to the most important metrics proposed for the same purpose in the literature. The metrics presented in this paper for evaluation and/or predicting class effort have been validated against several projects.

This paper is organized as follows. In Section 2, the metrics proposed for evaluating and predicting class effort on the basis of complexity/size are reported and compared with well-known metrics proposed in the literature. Then the validation of the most important metrics proposed is presented together with a comparison against metrics extracted from the literature in Section 3. This is followed by a very short overview of our metric framework and tool for automatic code metric evaluation (Section 4). Conclusions are drawn in Section 5.

2. Complexity/size of object-oriented systems

Most of the traditional *code metrics* proposed in the literature express the complexity/size as a function of a number of quantities which can be directly determined from system code – (McCabe, 1976; Henderson-Sellers and Edwards, 1990; Halstead, 1977); others adjust such measures with coefficients depending on the application domain. In addition, it has been often demonstrated that most of these metrics are related to the LOC which is a typical measure of code size (Shepperd and Ince, 1993).

Starting with the assumption that software complexity and size are strongly related to the effort value, even for object-oriented systems, several metrics have been defined (Henderson-Sellers et al., 1994; Thomas and Jacobson, 1989; Henderson-Sellers, 1991; Chidamber and Kemerer, 1994; Li and Henry, 1993; Henderson-Sellers, 1994; Hopkins, 1994; Laranjeira, 1990; Lorenz and Kidd, 1994). As discussed in the introduction, most of them consider only the functional aspects, while others are based on both structure (i.e., attributes) and functionality (i.e., methods) of class, or try to predict the final complexity by considering class public interface and definition in general (class FanIn and FanOut, i.e., method parameters). As is reported in the literature, among these, no one is considered fully satisfactory; for this reason, in our opinion, a more complete metric collection and an integrated metric framework for measuring all object-oriented aspects and, in particular, the system complexity/size are necessary.

In the following, several metrics related to software complexity and size are presented. As regards complexity/size three different levels have been considered – i.e., method, class and system. For each level, distinct metrics have been defined in terms of metrics of lower levels. The metrics proposed have also been compared with the most diffuse traditional metrics for complexity (McCabe Complexity, Mc (McCabe, 1976; Henderson-Sellers and Edwards, 1990)) and size (Halstead measure, Ha (Halstead, 1977) or LOC (Lorenz and Kidd, 1994)) estimation and with the most diffuse metrics for evaluating complexity/size of object-oriented systems (WMC (Chidamber and Kemerer, 1994) or Size2 (Li and Henry, 1993)). Traditional metrics have also been obviously adapted by the authors for dealing with object-oriented concepts when it was possible. The adaption corresponds to what has been done by many other authors in order to test the capability of traditional metrics in evaluating object-oriented systems. In general, the metrics presented can be regarded as a generalization of most of the object-oriented metrics proposed in the literature in the past.

In addition, the validation of the proposed class metrics shows the relevance of the metrics proposed with respect to the results obtained by using other metrics. The metrics proposed present a high number of components. This approach has been initially followed to analyze the importance of each metric term. In fact, a process of refinement has been performed in order to identify which of these terms is more significant than the others in modeling the actual effort; for example, by observing the influence of each metric component in estimating or predicting effort. This process has been supported by a statistic multilinear regression analysis (see Section 3).

In order to help the reader to understand the metric formulation and discussion, the authors have prepared from Table 1 in which the metrics and their corresponding meaning are reported in alphabetic order.

2.1. Method level metrics

At the method level, most of traditional metrics could be used since the method complexity is mainly due to functional aspects. For this reason, the number of LOC of a method/function (LOC), the Halstead measure of a method/function (Ha) (Halstead, 1977), and the McCabe complexity of a method/function (Mc) (McCabe, 1976; Henderson-Sellers and Edwards, 1990) can be used. On the other hand, data flow aspects related to method/function parameters are neglected (Zuse, 1991). In order to avoid the above problems of McCabe, LOC and Halstead metrics, more general metrics have been defined.

Therefore, a substantial improvement with respect to traditional pure functional metrics consists in considering also the complexity/size of the methods interface (list of parameters). The evaluation of complexity/size of method/function parameters can be even effective for considering cognitive aspects of method/function complexity (e.g., related to the cost of adoption and reuse) and for predicting costs of method/function implementation. Thus, in the context of OOP, this factor can be very useful for predicting the method complexity since the phase in which the class structure (attributes) and public interface (i.e., parameters of methods) are known. This usually happens much earlier with respect to method implementation (see Lorenz and Kidd, 1994), in the phases of detailed analysis.

For the above mentioned reasons, a generic method complexity can be defined

$$\mathrm{MC}_m = w_{\mathrm{MIC}_m}\mathrm{MIC}_m + w_m m, \qquad (1)$$

where MIC_m is the Method Interface Complexity/size, and m is a complexity/size metric for *method evaluation*; w_{MIC_m}, and w_m are weights, which obviously depend on metric m adopted. These weights are determined by means of the validation process reported in the following; they present different values in different phases of the software life-cycle for taking into account the changes in the incidence of functional and data flow aspects. Their trend can be evaluated by considering a set of similar projects. Their estimation can be performed in order to predict the costs of the next phases of life-cycle as well as for predicting the global cost of the system (with different confidence levels and reliability). Each specific set of weights has been obtained by using the corresponding

Table 1
Glossary of the metrics mentioned in this paper. Metrics with *m* parameter are evaluated on the basis of a functional metric selected from: MS, Mc, Ha or LOC; for example: CC_{Mc} Class Complexity/size based on McCabe ciclomatic Complexity

Metric	Comment
CC_m	Class complexity/size
CC'_m	Class complexity/size, predictive form
$CACI_m$	Class attribute complexity/size inherited
$CACI'_m$	Class attribute complexity/size inherited, predictive form
$CACL_m$	Class attribute complexity/size local
$CACL'_m$	Class attribute complexity/size local, predictive form
CMC_m	Class method complexity/size
$CMCI_m$	Class method complexity/size inherited
$CMCL_m$	Class method complexity/size local
$CMICI_m$	Class method interface complexity/size inherited
$CMICI'_m$	Class method interface complexity/size inherited, predictive form
$CMICL_m$	Class method interface complexity/size local
$CMICL'_m$	Class method interface complexity/size local, predictive form
CM_m	Class method complexity/size (pure functional)
CM_{LOC}	Class method size (Lorenz and Kidd, 1994)
CM_{Mc}	Class method McCabe complexity (McCabe, 1976; Henderson-Sellers and Edwards, 1990), see WMC
CI_m	Class method complexity/size inherited
CL_m	Class method complexity/size local
FC_m	Function complexity/size
GDC_m	Global definition complexity/size
GVC_m	Global variable complexity/size
Ha	Halstead metric (Halstead, 1977)
HSCC	Class complexity by Henderson-Sellers (1991)
LOC	Number of lines of code
Mc	McCabe ciclomatic complexity
MC_m	Method complexity/size
MIC_m	Method interface complexity/size
MS	Method size metric (Nesi and Querci, 1994)
NA	Number of attributes of a class
NAI	Number of attributes inherited of a class
NAL	Number of attributes locally defined of a class
NM	Number of methods of a class
NMI	Number of methods inherited of a class
NML	Number of methods local of a class
NAM	Number of attributes and methods of a class
NAMI	Number of attributes and methods inherited of a class
NAML	Number of attributes and methods locally defined of a class
NCL	Number of classes in the system
NGD	Number of global definitions
NGV	Number of global variables
NLC	Number of leaf classes in the system class tree
$NOOSC_m$	Non-object oriented system complexity
NRC	Number of root classes in the system class tree
NSF	Number of system functions/procedures
SC_m	System complexity/size
Size2	NA + NM: number of attributes and methods of a class (Li and Henry, 1993)
TJCC	Class complexity by Thomas et al. (1989)
T_m	Total *m*-based complexity/size of a system (pure functional)
WMC	Weighted methods for class (Chidamber and Kemerer 1994), CM_{Mc} according to our notation

effort, for example the total effort for obtaining the weight for the prediction of the total effort in the next projects.

The presence of MIC_m metric makes MC_m metric usable as a predictive metric, since MIC_m metric can be estimated even if the method has not been implemented, but its prototype is known. MIC_m takes into account the usually neglected relationships which are established among classes by means of the parameters of the methods. MIC_m is estimated by considering the sum of class complexity/size of each method parameter (the infinite recursion in the estimation of class complexity/size is constrained by limiting the recursion to one level). Elementary types of the language have an assigned complexity. For these reasons MIC_m and MC_m are greater than zero when only the class definitions in terms of attributes and method prototypes are present even partially.

Metric m can be the well-known McCabe ciclomatic complexity (McCabe, 1976; Henderson-Sellers and Edwards, 1990) (i.e., $m = \mathrm{Mc}$), the number of LOC (i.e., $m = \mathrm{LOC}$), the Halstead metric (Halstead, 1977) (i.e., $m = \mathrm{Ha}$), or our Method Size metric (Nesi and Querci, 1994) (i.e., $m = \mathrm{MS}$). Therefore, a set of structurally similar metrics ($\mathrm{MC}_{\mathrm{Mc}}$, $\mathrm{MC}_{\mathrm{LOC}}$, $\mathrm{MC}_{\mathrm{Ha}}$, $\mathrm{MC}_{\mathrm{MS}}$) have been defined. In this paper, the term "complexity/size" metric is adopted, since some of the above metrics evaluate program complexity and the others estimate program size, then the parametrized metrics can be either a complexity or a size metric. MC_m metrics are more complete than the simple complexity/size metrics on which they are based. In fact, they present the terms MIC_m to adjust the traditional functional metrics for both: (i) to take also into account complexities due to data flow, (ii) to obtain measures even when the method is only partially implemented or simply identified. The above metrics can be obviously useful for obtaining more complete evaluations of non-object-oriented systems.

MS metric has been defined by the authors in order to obtain a reliable evaluation of method size (Nesi and Querci, 1994). This metric partially avoids the problems of McCabe complexity which in some cases estimates low values for synthetic constructs of the language which once they are exploded produce a high complexity. MS is based on code analysis and takes into account the presence of elementary language tokens and constructs in both class definition and method implementation (`if`, `for`, `switch`, `else`, `continue`, `break`, `return`, `while`, `public`, `private`, `do`, `case`, `typedef`, `struct`, `default`, `class`, `friend`, `virtual`, `int`, `char`, `float`, `etc.`, assignments, function/method prototype/definition, operator definition, casting, and algebraic, comparative, and Boolean operators) and the nesting of levels (Nesi and Querci, 1994). Since this metrics considers both functional and definitional aspects it results better ranked for evaluating the above mentioned metrics.

2.2. Class level metrics

At the level of class, by using McCabe, Halstead, and LOC metrics it is immediately possible to define a set of pure functional class metrics. Their usual definition for working with classes is obtained by considering the sum of complexities of all class methods:

$$\mathrm{CM}_m = \sum_{i}^{\mathrm{NM}} m(i). \tag{2}$$

where CM_m is the complexity/size metric for class methods obtained by using metric m, NM is the number of class methods. Therefore, the following class metrics can be defined: $\mathrm{CM}_{\mathrm{Mc}}$, a class level metric based on McCabe metric; $\mathrm{CM}_{\mathrm{Ha}}$, a class level metric based on Halstead metric and $\mathrm{CM}_{\mathrm{LOC}}$, a class level metric based on the number of LOC. In the literature, it has been often demonstrated that CM_m metrics are not very suitable for evaluating object-oriented projects, since they are not capable of considering the object-oriented aspects (Thomas and Jacobson, 1989; Henderson-Sellers, 1991). In fact, they neglect information about class specialization (is_a, that means code and structure reuse), and class association and aggregation (is_part_of and is_referred_by, that mean class/system structure definition and dynamic managing of object sets, respectively). On the other hand, $\mathrm{CM}_{\mathrm{Mc}}$ (equivalent to WMC (Chidamber and Kemerer, 1994)) and $\mathrm{CM}_{\mathrm{LOC}}$ (used in Lorenz and Kidd, 1994) have been adopted as good compromises between precision and simplicity of evaluation. $\mathrm{CM}_{\mathrm{LOC}}$ can be set to count also the lines devoted to the definition of classes.

On the other hand, on the basis of the metrics presented in Section 2.1 (see Eq. (1)), a more general metric for evaluating *class complexity/size due to methods* (functional part of the class) has been defined

$$\mathrm{CMC}_m = \sum_{i}^{\mathrm{NM}} \mathrm{MC}_m(i), \tag{3}$$

where CMC_m (Class Method Complexity) is the whole complexity/size due to methods for a class estimated on the basis of metric m. Therefore, the following class level metrics have been defined: $\mathrm{CMC}_{\mathrm{Mc}}$, based on McCabe metric; $\mathrm{CMC}_{\mathrm{Ha}}$, based on Halstead metric; $\mathrm{CMC}_{\mathrm{LOC}}$, based on the number of LOC, and $\mathrm{CMC}_{\mathrm{MS}}$, based on our MS metric. This last set of metrics takes also into account the method interface according to Eq. (1)

$$\mathrm{CMC}_m = w_{\mathrm{MIC}_m} \sum_{i}^{\mathrm{NM}} \mathrm{MIC}_m(i) + w_m \sum_{i}^{\mathrm{NM}} m(i). \tag{4}$$

These metrics are capable of obtaining estimation values even when the class implementation is not yet available if the class definition in terms of attributes and method prototypes are known, even partially. This is due to the presence of MIC_m as explained in the previous section. Even these metrics are not capable of considering the typical relationships of specialization, aggregation and association of object-oriented.

According to the above discussion, a fully object-oriented metric for evaluating class complexity/size has to consider also attributes and methods both locally defined and inherited. These factors must be considered for evaluating the cost/gain of inheritance, and that of the other relationships. Therefore, the class complexity, CC, is regarded as the weighted sum of local class complexity (CCL) and inherited class complexity (CCI) (recursively till the roots are reached)

$$\mathrm{CC} = \mathrm{CCL} + \mathrm{CCI}. \tag{5}$$

Since CCL and CCI are expressed in terms of complexities due to attributes and methods (local and inherited), respectively, CC assumes the form

$$CC = w_{CACL}CACL + w_{CMCL}CMCL + w_{CACI}CACI + w_{CMCI}CMCI, \quad (6)$$

where CACL is the Class Attribute Complexity Local, CMCL the Class Method Complexity Local, CACI the Class Attribute Complexity Inherited, CMCI the Class Method Complexity Inherited (e.g., complexity reused). In this way, CC takes into account both structural (attributes, relationships of is_part_of and is_referred_by) and functional/behavioral (methods, method "cohesion" by means of CMICL and CMICI included in CMCI and CMCL, respectively) aspects of class. Weights in Eq. (6) must be evaluated by a multilinear regression analysis by knowing class effort Rousseeuw and Leroy, 1987), as will be discussed in the following. In general, w_{CACI} is typically negative stating that the inheritance of attributes leads to save complexity/size and, thus, effort. Since the effort for defining and implementing overwritten and overloaded class members is a real cost these are evaluated in both local and inherited classes. The weights take into account also these aspects dependently on the usual style of the company/team and on the context of the application.

In CC, CACL and CMCL are defined as follows:

$$CACL = \sum_{i}^{NAL} AC_i, \quad (7)$$

$$CMCL = \sum_{i}^{NML} MC(i), \quad (8)$$

where NAL, Number of Attributes Local; NML, Number of Methods Local. CACI and CMCI are analogously defined (see Table 1 for a summary). Please note that attributes can be: (i) class instances and, thus, they are evaluated by considering metric CC of their corresponding class (i.e., AC = CC), or (ii) basic types (e.g., `char`, `int`, `float`, etc.) for which the complexity is posed to predefined values. The same mechanisms are used for estimating CACI and CMCI from superclasses. It should be noted that MC can be one of the previously presented metrics at the level of method, see Eq. (1). Therefore, CMCL and CMCI are estimated on the basis of Eq. (4). In particular, by considering Eqs. (4) and (6):

$$CC_m = w_{CACL_m}CACL_m + CMCL_m + w_{CACI_m}CACI_m + CMCI_m, \quad (9)$$

where

$$CMCL_m = w_{CMICL_m}CMICL_m + w_{CL_m}CL_m, \quad (10)$$

$$CMCI_m = w_{CMICI_m}CMICI_m + w_{CI_m}CI_m. \quad (11)$$

Note that, attribute complexity/size of class instances are also considered by using CC_m; then, their weights depend on the kind of metric used for evaluating methods. On the basis of the above discussion, several fully object-oriented metrics based on functional metrics, CC_{Mc}, CC_{Ha}, CC_{LOC} and CC_{MS}, have been defined.

During the analysis and validation of these metrics other versions of them have been adopted and tested. These versions have been obtained by starting from the previously defined metrics and reducing the number of terms – for example, by considering only methods, only attributes, only members inherited, only local members, etc. This analysis has been performed on the basis of technique (Rousseeuw and Leroy, 1987) for identifying the most important terms.

It should be noted that values for CC_m metrics are obtained even if only the class structure (attribute and method interface) is available. This can be very useful for class evaluation and prediction since the early phase of class life-cycle. The weights or the interpretation scale can be adjusted according to the phase of the system life-cycle in which they are evaluated as in Nesi and Campanai (1996).

Please note that the WMC of Chidamber and Kemerer (1994) is the sum of all McCabe complexities of class-methods which is equivalent to the previously mentioned CM_{Mc}. It is also very similar to CL which is estimated by using Mc previously shown. The metric proposed by Thomas and Jacobson (1989) and the evolution proposed by Henderson-Sellers (1991) can be defined in terms of CC components, that is, $TJCC = w_{CACL}CACL + w_{CL}CL$ and $HSCC = w_{CACL}CACL + w_{CL}CL + w_{CI}CI$, respectively. For these reasons, CC_m can be considered as a generalization of these metrics.

As previously pointed out, metric CC_m can also be used for predicting class complexity/size. In particular, the prediction can be obtained on the basis of class definition, that is, attributes declarations and methods prototypes. This estimation can be performed during system analysis/early-design. According to these requirements the following predictive version of CC_m metrics has been obtained:

$$CC'_m = w'_{CACL_m}CACL'_m + w'_{CMICL_m}CMICL'_m + w'_{CACI_m}CACI'_m + w'_{CMICI_m}CMICI'_m. \quad (12)$$

In this case, $CACI'_m$ and $CACL'_m$ are obviously estimated on the basis of the CC'_m of class members. Even in this case, the weights must be evaluated on the basis of a set of reference projects by using a validation process such as that reported in the next section.

A cheaper approach can be simply based on counting the number of local attributes and methods (see metric Size2 = NAL + NML defined by Li and Henry (1993)). On the other hand, the simple counting of class members (attributes and methods) could be in many cases too coarse. For example, when an attribute is an instance of a very complex class its presence in a class often implies a high cost of method development which is not considered simply by increasing NAL of one unit. Moreover, Size2 does not consider the class members inherited (that is, reuse). For these reasons, in order to improve the metric precision, a more general metric has been defined by considering the sum of the number of class attributes and methods both local defined and inherited, respectively.

$$\mathrm{NAM} = \mathrm{NAML} + \mathrm{NAMI}, \tag{13}$$

therefore, NAM can be expanded assuming the form

$$\mathrm{NAM} = w_{\mathrm{NAL}}\mathrm{NAL} + w_{\mathrm{NML}}\mathrm{NML} + w_{\mathrm{NAI}}\mathrm{NAI} + w_{\mathrm{NMI}}\mathrm{NMI}. \tag{14}$$

Also in this case, the typical values of weights must be estimated by using a multilinear regression technique.

Therefore, code metrics can be used for effort prediction because according to OOP some definitions about classes are available since the early phases of the software life-cycle. For example, during the analysis one has at least, for each class, the number of attributes and methods (not the definitive number but an early approximation), then the types of the attributes and the parameters of methods are defined (even partially), then class relationships are defined, then the phases of method implementation begins, etc. Therefore, specific code metrics based on the number of class members, attributes, methods and definitions can be profitably used as metrics for predicting costs of the successive phases as well as for predicting the final cost.

2.3. System level metrics

According to the current C++ interpretation of the OOP, the *system level* can be comprised of: (i) a set of classes which can be organized in one or several class trees, (ii) a set of C functions/procedures (even the main program in C++ is a function/procedure, if no window support is present), (iii) a set of global definitions of types, structures, unions, etc., and (iv) a set of global declarations of variables (static instantiation of global variables). Differently from what is stated by the OOP, in some C++ systems these factors are a significant part of the system itself. The reference projects used for the metric validation reported in the following present so small components of type (ii), (iii) and (iv) which could be even neglected. On the other hand, in other C++ systems these non-object-oriented parts may be too relevant to be neglected.

At the system level, several direct metrics can be defined in order to analyze the system structure – e.g.: NCL, Number of CLasses in the system; NSF, Number of System Functions/procedures; NGD, Number of Global Definitions (excluded classes, only `typedef`, `enum` and `struct`); NGV, Number of Global Variables (global declaration of instances); etc.

On the other hand, System Complexity, SC_m, is an indirect measure which in our framework is defined as

$$\mathrm{SC}_m = w_{\mathrm{CC}_m}\sum_i^{\mathrm{NCL}}\mathrm{CC}_m(i) + w_{\mathrm{FC}_m}\sum_i^{\mathrm{NSF}}\mathrm{FC}_m(i) + w_{\mathrm{GDC}_m}\sum_i^{\mathrm{NGD}}\mathrm{GDC}_m(i) + w_{\mathrm{GVC}_m}\sum_i^{\mathrm{NGV}}\mathrm{GVC}_m(i), \tag{15}$$

where FC_m, GDC_m and GVC_m are complexity/size metrics for taking into account non-object-oriented aspects, that is, functions, global definitions and global declaration of variables, respectively. Definitions and variable declarations are counted by using the same mechanisms as that adopted for classes and method parameters, respectively. Note that FC_m is evaluated by applying the previously defined metric CM_m (see Eq. (2)) to C functions/procedures. SC_m is computed by using the complexity of all system classes augmented by functional and data complexities due to global parts which can be found in object-oriented programs written in C++. In SC_m, weights w_{CC_m}, and w_{FC_m} are typically set to 1, while w_{GDC_m} and w_{GVC_m} have a lower value, typically 1/20 (these values have also been estimated by means of a multilinear regression analysis). This fact does not mean that the adoption of global variables makes the system less complex or cheaper, but that GVC_m and GDC_m present a different scale with respect to the other metrics. It should be noted that since SC_m has been defined in terms of CC_m which, in turn, can be estimated since the early phases of class life-cycle (CC'_m), the system complexity presents the same capability.

SC_m metric is capable of producing a more precise evaluation of system complexity since it considers related to the aspects of functional, data and OOP relationships, differently from traditional system metrics: Total McCabe Complexity (T_{Mc}) indirectly based on McCabe's metric, the Total Halstead T_{Ha}), and the Total Number of Lines Of Code (T_{LOC}) defined as on the basis of previously defined metrics

$$T_m = \sum_i^{\mathrm{NCL}}\mathrm{CM}_m(i) + \sum_i^{\mathrm{NSF}}\mathrm{FC}_m(i), \tag{16}$$

where also in this case m can be Mc, Ha, MS or LOC.

3. Comparison and validation of metrics

A metric analysis has been performed in order to identify which metric of the above-mentioned class metrics is better ranked for evaluating and/or predicting class effort.

The comparative analysis with validation has been carried out among the previously defined metrics and those already defined in the literature. Moreover, since most of the new metrics defined are very complex and computationally expensive to be evaluated (see

Eq. (9)), an analysis to verify the influence of their parameters in producing the final result has been performed in order to identify the minimum number of parameters which are needed to obtain reliable measures. Therefore, the analysis performed is more than a simple validation since it has produced a clear view of the above cited and defined metrics for estimating class effort. The formal definition of the computational complexity is quite hard since it depends on the relationships defined among classes.

The validation has been performed by considering three projects with the overall values reported in Table 2 (where: NRC is the Number of Root Classes, NLC the number of leaf classes, and $NOOSC_{MS}$ is the system complexity due to non-object-oriented parts, see Eq. (15)). The first reference project is an object-oriented CASE tool TOOMS (Bucci et al., 1993, 1994); the second is QV, an object-oriented class library for building applications having GUI on Motif environment, and the third project is the object-oriented software for configuring and controlling automatic milling machines, called ICOOMM (ELEXA, CB-Ferrari, Italy). In both TOOMS and ICOOMM projects, class libraries for GUI (i.e., CommonView and MSVC++, respectively) have been used.

All the above projects have been built by using object-oriented analysis and design methodologies (i.e., Booch, 1994), maintaining under control the development process by using the metrics presented above with the tool discussed in Section 4. During the development process periodic evaluation sections have been performed; moreover, the actual effort for each class has been recorded on the project database. At the end of the projects, the measures obtained on the several versions of the projects and the actual effort recorded have been used by a multilinear regression analysis for evaluating weights and, thus, for obtaining specific metrics for the different phases of the software development.

According to the values reported in Table 2, the mean number of methods per class is 8.65, the mean number of class per tree is 5.8. These and others values are in accordance with the typical values obtained by others Lorenz and Kidd, 1994; Henderson-Sellers et al., 1994) when criteria of object-oriented analysis and design are used by skilled people.

In this paper, only the validation at the level of class has been reported since it can be considered the most interesting one for the OOP. On the other hand, we have also proposed method level metrics since they are used for defining class level metrics, and system level metrics to give an idea of project dimensions. Note that, being available the effort for each class the metric validation has been based on 524 measurements. As demonstrated by the confidence values reported from the multilinear regression analysis a confident validation has been obtained even with only 47300 LOC. Moreover, the identified weights evaluated for the discussed metrics have been used to maintain other projects under continuous metrication, confirming that the values obtained are reliable. In their evaluation sections, predictive and a-posteriori metrics have been used to predict the cost of the next phase and evaluate the current costs. The simultaneous evaluation of actual cost of these projects has allowed the revalidation of the metrics with their weights, which have been confirmed in value and sign.

As previously discussed, the above mentioned metrics can be classified in *a-posteriori* and predictive metrics. The validation/analysis of metrics for these two categories are separately discussed in Sections 3.1 and 3.2. In other terms, a-posteriori *or code metrics* take into account the whole class aspects, attributes and methods (locally defined and inherited) and need for their evaluation the complete C++ code of the system. *Predictive metrics* are those which typically count the number of class members or are capable of obtaining estimations since the phase in which the class is defined and not yet implemented. By using the effort data (in man-hours) relative to the above mentioned projects, the optimal weights which must be placed into the expressions of CC_{MS}, CC_{Mc}, CC_{Ha}, CC_{LOC}, CC'_{MS}, CC'_{Mc}, CC'_{Ha}, CC'_{LOC}, and NAM, to obtain the maximum correlations have been estimated. This analysis has been performed by using a multilinear least-squares technique (Rousseeuw and Leroy, 1987) considering: (i) the relationship between effort and metrics as linear, (ii) the effort as the value of metrics, and (iii) the weights as unknowns.

3.1. A-posteriori estimation

In Table 3, the results of multilinear regression analyses are reported. The analyses have been carried out by considering the real effort in man-hours and metrics: CC_{MS}, CC_{Mc}, CC_{LOC} and CC_{Ha}, by using the techniques discussed in Rousseeuw and Leroy (1987).

In Table 2, the values of correlation, the variance of correlation (variance of the error function between the metric and the actual effort), the scale of the regression line evaluated, and other statistical values are reported for each CC_m metric. Hence, it can be observed that a high value of correlation has been obtained for all met-

Table 2
Overall values for the projects used for the estimation of weights

	NCL	NRC	NLC	NM	SC_{MS}	$NOOSC_{MS}$	T_{LOC}	T_{Mc}	T_{Ha}
Total	524	90	368	4537	233935	1230	47337	10086	36188534

Table 3
Results of the multilinear regression analysis for *effort evaluation* of classes by using metrics CC_m: values of weights and their corresponding confidence values are reported

CC_m	$m = MS$			$m = Mc$			$m = LOC$			$m = Ha$		
	w	t-value	p-value	w	t-value	p-value	w	t-value	p-value	w	t-value	p-value
$CACL_m$	.001	4.97	.000	.001	4.39	.000	.001	4.82	.000	.001	4.19	.000
CL_m	.022	10.01	.000	.092	7.46	.000	.016	8.63	.000	.006	8.37	.000
$CMICL_m$	.042	3.39	.001	.042	2.66	.009	.053	4.10	.000	.087	8.18	.000
$CACI_m$	−.026	−3.48	.001	−.070	−3.57	.001	−.023	−2.49	.014	−.014	−1.95	.054
CI_m	−.002	−.72	.471	.014	1.28	.202	−.001	−.59	.554	.001	.53	.598
$CMICI_m$	.013	1.75	.083	.014	1.39	.166	.010	1.49	.140	.002	.28	.776
Eff-corr.		0.945			0.926			0.935			0.937	
Variance		192.98			149.28			145.68			216.63	
LS scale		2.928			3.379			3.166			3.120	
R-squared		0.911			0.881			0.896			0.899	
F-stat		159.24			115.79			134.00			138.47	
p-value		0.000			0.000			0.000			0.000	

rics. The correlation between CC_{LOC} and effort presents the lowest value of variance of correlation. Lower values of variance correspond to less-spread distributions of the departure from the model – i.e., a lower probability to get the wrong effort estimation or prediction by using the selected metric. For these reasons, the analysis reported shows that traditional functional metrics can be profitably employed for evaluating object-oriented systems if they are used as a basis for more complete metrics – that is, by using them for defining metrics capable of considering all aspects of the OOP. This is also confirmed by the results presented in Table 4, where the well-known metrics for class estimation: WMC, the pure functional metrics CM_{LOC}, and CM_{Ha} are also compared on the basis of correlation value and variance of the departure from the model. Please note that WMC is widely diffuse for evaluating object-oriented systems, but it consists in evaluating the McCabe complexity of class methods, that is CM_{Mc}. The metrics proposed in this paper present both a higher value of correlation and a lower value of variance; thus, they are better ranked with respect to those reported in Table 4. The differences among the values of correlation are not so strong but, the values of variance are in these cases more important since they give an idea of the estimation confidence.

Table 3 reports the values of the weights, w, with their corresponding *t-values* and *p-values* Rousseeuw and Leroy, 1987). Intuitively, t-value is an index which establishes the importance of a coefficient for the general model. A coefficient can be considered significant if t-value is greater than 1.5 (since a high number of measures have been used for the regression). On the basis of t-value, the confidence intervals can be evaluated. p-value can be considered a probability; when it is less than 0.05 the corresponding coefficient is significant with a confidence of 5%. Therefore, components CI_m and $CMICI_m$ are the least significant in all CC_m metrics. In fact, by removing these coefficients in the definition of CC_m metrics, very similar results are obtained with a simpler metric. In our experiments, it was shown that coefficients $CACL_m$, CL_m, and $CMICL_m$ are enough for obtaining correlations of 0.93 (high values of t-values and p-values all lower than 0.05). By removing terms of metric CC_m (see Eq. (9)), a very light decrease in correlation and an increment of the variance have been obtained (until the above mentioned version with three terms was defined); while a strong degradation was obtained by removing one of the remaining terms.

The process of analysis has also demonstrated that the values which can be obtained by using TJCC and HSCC metrics are less satisfactory with respect to those obtained by CC_m which result to be their generalization. For these metrics, values of correlation lower than those obtained for CC_m have been registered. In particular, HSCC metric presents CI which has been demonstrated to be totally un-useful for effort estimation.

By considering the weight values estimated for metrics CC_{MS}, CC_{Mc}, CC_{LOC} and CC_{Ha}, it can be noted that weight related to components evaluating complexity/size of attributes and methods inherited are negative for

Table 4
Comparison between the metric defined for effort evaluation of classes with respect to the most important metrics of the literature

	$WMC = CM_{Mc}$ (Chidamber and Kemerer, 1994)	CM_{LOC} (Lorenz and Kidd, 1994)	CM_{Ha} (Halstead, 1977)
Eff-corr.	0.90	0.91	0.82
Variance	245	186	423

most of those metrics. This confirms that the inheritance mechanism is a means for effort saving.

A ratio of about $\frac{1}{10}$ for $w_{\mathrm{CI}}/w_{\mathrm{CL}}$, as was estimated in our reference projects, is in compliance with the main concepts of object-oriented analysis and design. In fact, this ratio states that an opposite ratio is present between method complexity/size: CI/CL. Thus, a typical class inherits methods for a complexity 10 times bigger with respect to that locally defined. Similar values have been obtained by several other authors as previously stated.

3.2. Predictive estimation

In Table 5, the results of the multilinear regression analysis by considering the real effort and metrics $\mathrm{CC}'_{\mathrm{MS}}$, and $\mathrm{CC}'_{\mathrm{LOC}}$, by using the technique discussed in Rousseeuw and Leroy (1987), are reported. Also in this case, the table reports the values of the weights, w, with t-values and p-values. As for the a-posteriori metrics the weights of the coefficients of complexity/size of attributes inherited are negative, confirming effort saving by using the inheritance (as above, the effort saving is comprehensive of the cost of reuse, thus, globally, it is better to inherit rather than to rebuild). Component CMICI_m is the lowest significant in all metrics. In fact, by removing this term in the definition of CC'_m metrics (and re-evaluating the multilinear regression line) the corresponding correlation remains close to 0.87, while the variance decreases to about 200. In this case, the reduction of terms improves the results obtained with the complete version of CC'_m reported in Table 5, where variances in the range 600–700 are obtained. Please note that the absolute value of weights also depends of the phase of the development life-cycle in which the prediction is performed and if the prediction is carried out for evaluating the effort to reach the next phase or to complete the project.

The multilinear regression analysis has also been employed for analyzing the importance of the coefficients of metric NAM (see Eq. (14)). For the version of NAM with all its terms a correlation of 0.717, a variance of about 3000, and a scale of 6.2, have been obtained. In this case, the fact that w_{NMI} is negative means that inheriting methods are less expensive than defining them exnovo. In out reference projects a ratio of 5 between the number of methods inherited and locally defined was estimated. This is in compliance with the values obtained by others when the good criteria for object-oriented analysis and design were employed by skilled people.

Please note that a correlation value of 0.719 with 1700 of variance has been obtained for Size2. In this case, it has been observed that the most significant components of NAM are NAL and NML, while NAI and NMI are only partially correlated with the effort (p-values equal to 0.32 and 0.33, respectively), thus confirming the validity of metric Size2 defined in Li and Henry (1993). Please note that, when independent variables with very high p-values and very low t-values (low correlation with the effort) are removed, then the correlation and its variance may usually improve (Rousseeuw and Leroy, 1987).

The values of correlation and variance for Size2 and NAM are less satisfactory with respect to predictive metrics CC'_m; on the other hand, Size2 metrics is more easily evaluated. For these reasons, the suggestion is to adopt Size2 when the number of methods and attributes is known while it is better to consider CC'_m when more information on class definition is available. CC'_m can be used even when class attribute types and class method prototypes are only partially known. In these conditions, each method/attribute identified but not fully specified as type/prototype can be simply counted as a value equal to the metric scale (see Table 5).

This method can be very encouraging because the cost of obtaining values for these metrics since the early phases of software life-cycle is very low. Therefore, small errors should be accepted in the predictive estimation of effort by using those metrics.

Table 5
Results of the multilinear regression analysis for *effort prediction* of classes by using metrics CC'_m: values of weights and their corresponding confidence values are also reported

CC'_m	m = MS			m = LOC		
	w	t-value	p-value	w	t-value	p-value
CACL'_m	0.002	9.10	0.000	0.002	9.05	0.000
CMICL'_m	0.123	9.33	0.000	0.123	9.29	0.000
CACI'_m	−.027	−2.48	0.015	−.028	−2.31	0.023
CMICI'_m	0.010	1.56	0.121	0.009	1.44	0.152
Eff-corr.		0.881			0.880	
Variance		688.98			770.55	
LS scale		4.230			4.243	
R-squared		0.810			0.809	
F-statistic		101.87			101.19	
p-value		.000			.000	

3.3. Context

The values for the weights depend on the application context. Therefore, it is possible to obtain more precise results by adjusting the weights depending on the type of the system under development. This can be simply done by using 2–3 reference projects into the selected area and estimating weights with the previously applied method. The reference projects must be in compliance with the OOP and the quality profile defined on the basis of company's needs (Nesi and Campanai, 1996). On the other hand, the magnitude of weights remain quite unchanged (among the weights only w_{CMICL} and w_{CACI} tend to increase their values along the development life-cycle). The reported weights are valid for applications which adopt an object-oriented library of classes for managing GUI with windows (e.g., Motif plus CommonView, MS-Windows plus MSVC++). They have approximately the 30% of code devoted to user interface management.

3.4. System level

It should be noted that by using the weights estimated several other small-medium sized projects (with similar characteristics) have been measured with satisfactory results. In particular, in the case of LIOO project (implementation of a graphical and interactive editor for music) it has been early analyzed for predicting effort and, during its implementation, for periodically verifying costs of development (in effect, the project has been controlled during its life-cycle, in order to maintain its costs and quality acceptable according to the quality profile defined, see Nesi and Campanai (1996)).

On the basis of the estimation of NAML, after the analysis/design phases (when only an effort of 765 man-hours was spent), a final effort of 980 was predicted. This prediction has been confirmed by the final version of the LIOO system which consists of 104 classes, 12843 lines of code was built by using 1150 man-hours (for analysis, design, code and test, plus other 639 hours for documenting); a team of seven people with a project manager/analyst was employed. As regards the final SC_{MS}, a value of 389 man-hours (only for coding) was estimated, while the real effort was 383.5 man-hours. The estimations on final code have confirmed the efficacy of the weights evaluated and, thus, the power of the metrics defined as depicted in Fig. 1, where the correlation of CC_{MS}-Effort is equal to 0.93 with variance of 95, while the correlation of WMC-Effort equal to 0.89 and variance of 180, etc.

In addition, since the metrics at the system level are mainly based on class complexity plus few functional parts, it is quite obvious that even at the system level the object-oriented metrics proposed (CC_m, CC'_m, NAM) are better ranked with respect to system metrics defined in terms of WMC (Chidamber and Kemerer, 1994), CM_{Ha} (Halstead, 1977), Size2 (Li and Henry, 1993) and CM_{LOC} (Lorenz and Kidd, 1994) metrics. This assumption is quite true since according to the OOP the system is substantially regarded as a collection of class-

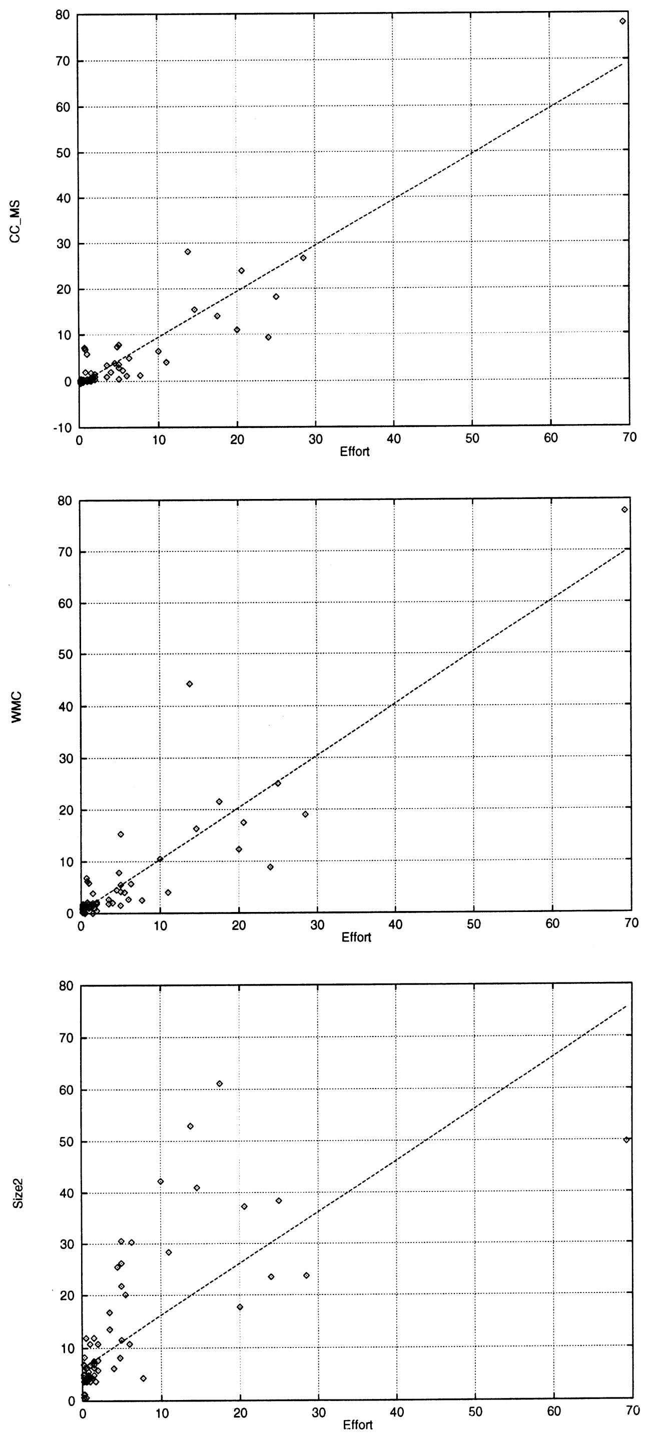


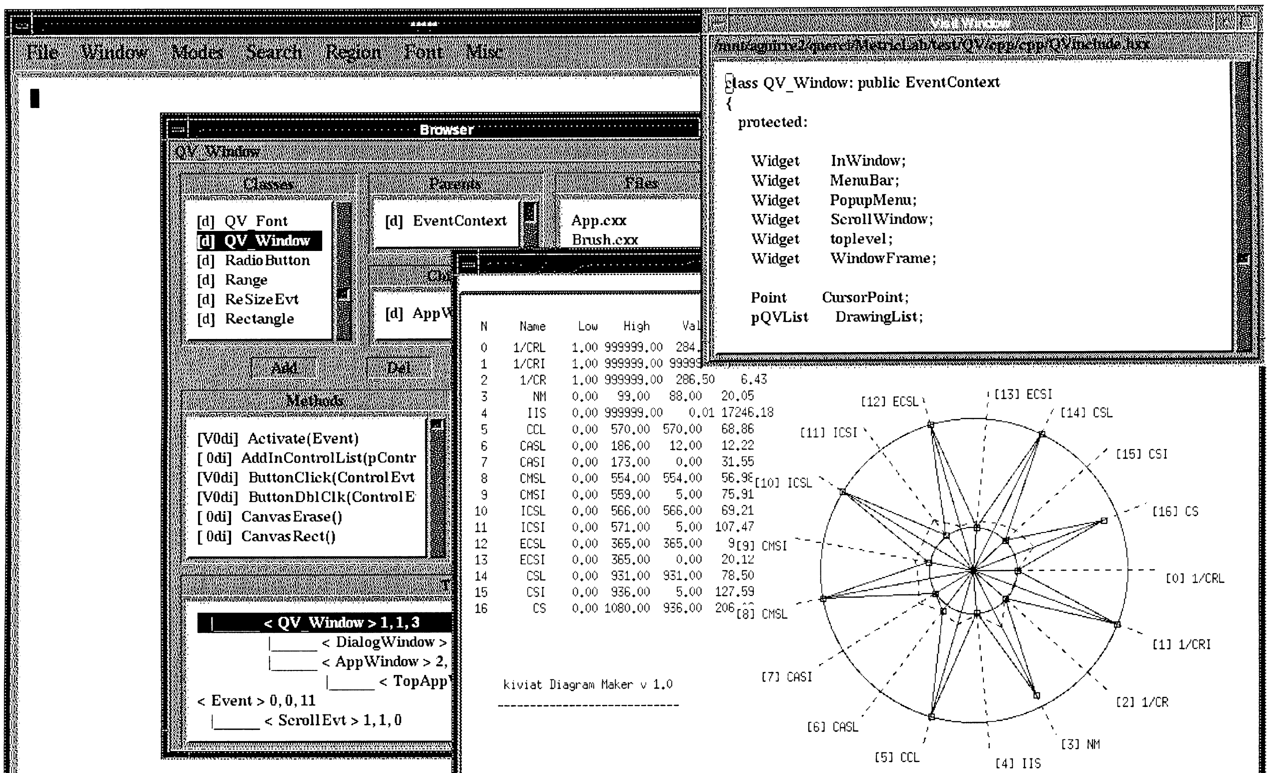
Fig. 1. Correlation of metrics with respect to effort, from left to right CC_{MS}, WMC, Size2 (evaluated on project LIOO, normalized values). Line with the optimal correlation is also reported.

es. On the contrary, most of the newly proposed metrics are computationally more complex.

4. Short overview of metric framework

In the literature, many metrics frameworks have been presented – e.g., (Henderson-Sellers, 1991; Zuse, 1994; Meyer, 1990; Henderson-Sellers, 1993; Coulange and Roan, 1993; Li and Henry, 1993; Brito e Abreu et al., 1995). The metrics previously proposed belong to a metric framework such as many other metrics. Our approach is based on three different system views: a *technical*, a *cognitive*, and a *process-oriented* view (Nesi and Campanai, 1996). The *technical* view refers to the software engineering aspects of system specification (size, complexity, etc.); the *cognitive* view takes into account the external understandability and verifiability of system components and libraries, and the process-oriented view refers to system aspects which are influenced by or can influence the process of system development (productivity, reuse, size, cost, etc.). Each metric that belongs to this view can also be used for producing predictions on its corresponding measure. These three views are evaluated in a common measurement framework in which each view can influence the others as in Nesi and Campanai (1996). Metrics of each view can be employed in different phases of system evolution: the cognitive metrics during system development and/or in system maintenance, the technical metrics for the evaluation/certification of some specific characteristics of the system; the process-oriented metrics for evaluating the impact of technology on the whole development process. All metrics have been analyzed and validated by using the previously presented technique.

All these metrics can be evaluated by using the tool, Tool for Analyzing C++ code (TAC++) specifically defined for C++ language (see Fig. 2). The main aspects of our metric framework have been inherited from that presented in Campanai and Nesi (1994) and Nesi and Campanai (1996) by the same research group; Nesi and Campanai (1996) version worked on formal object-oriented language named TROL (Bucci et al., 1994) and was directly integrated into its CASE tool. An accurate work has been performed for remapping metrics from TROL to C/C++ and for building a new and independent software instrument for obtaining automatic evaluation of more that 130 direct and indirect metrics at the level of class. TAC++ is a research prototype and not a commercial tool. It is very suitable for



```
[d] QV_Font
[d] QV_Window
[d] RadioButton
[d] Range
[d] ReSizeEvt
[d] Rectangle

[d] EventContext

App.cxx
Brush.cxx

[V0di] Activate(Event)
[ 0di] AddInControlList(pContr
[V0di] ButtonClick(ControlEvt
[V0di] ButtonDblClk(ControlE
[ 0di] CanvasErase()
[ 0di] CanvasRect()

|______ < QV_Window > 1, 1, 3
        |______ < DialogWindow >
        |______ < AppWindow > 2,
                |______ < TopApp
< Event > 0, 0, 11
  |______ < ScrollEvt > 1, 1, 0
```

```
 N    Name    Low      High       Val
 0   1/CRL   1.00 999999.00   284.
 1   1/CRI   1.00 999999.00 99999
 2    1/CR   1.00 999999.00   286.50     6.43
 3      NM   0.00     99.00    88.00    20.05
 4     IIS   0.00 999999.00     0.01 17246.18
 5     CCL   0.00    570.00   570.00    68.86
 6    CASL   0.00    186.00    12.00    12.22
 7    CASI   0.00    173.00     0.00    31.55
 8    CMSL   0.00    554.00   554.00    56.9
 9    CMSI   0.00    559.00     5.00    75.91
10    ICSL   0.00    566.00   566.00    69.21
11    ICSI   0.00    571.00     5.00   107.47
12    ECSL   0.00    365.00   365.00     9
13    ECSI   0.00    365.00     0.00    20.12
14     CSL   0.00    931.00   931.00    78.50
15     CSI   0.00    936.00     5.00   127.59
16      CS   0.00   1080.00   936.00   206

  kiviat Diagram Maker v 1.0
  ----------------------------
```

```
/mnt/agnirre2/querci/MetricLab/test/QV/cpp/cpp/QVinclude.hxx

class QV_Window: public EventContext
{
  protected:

    Widget     InWindow;
    Widget     MenuBar;
    Widget     PopupMenu;
    Widget     ScrollWindow;
    Widget     toplevel;
    Widget     WindowFrame;

    Point      CursorPoint;
    pQVList    DrawingList;
```

Fig. 2. TAC++ at work.

studying metric behavior and includes class browser and editing capabilities; therefore, it can be considered an integrated instrument for developing and maintaining under control (e.g., quality, effort, etc.) C/C++ code. This is also guaranteed by the fact that most of the defined metrics are capable of producing results since the early phases of the software life-cycle. The aim of our framework and tool is to cover the entire development life-cycle by using a unique tool in order to provide to software developers and managers an appropriate instrument for automatically analysing software since the early phases of system development when only classes and their main relationships are defined.

TAC++ is capable of performing measures in compliance with ISO 9126 since specific metrics for the classical six features of quality profile can be easily evaluated. TAC++ allows the users to define new high-level metrics on the basis of metrics, indexes, and variables already defined in the metric framework by connecting them with classical operators. In addition, a dedicated and fully configurable visualiser can be used for defining specific graphical representations of views/profiles (by using Kiviat, line graphs, bars, pies, etc.). The views defined can be used for monitoring aspects of the system under assessment, at level of class, method and system on the basis of the context for which they are defined (quality analysis, reuse analysis, verifiability and test analysis, etc.). In these graphical views of profiles, different weights, reference thresholds, minimum and maximum values for diagrams can be set according to the company/user goals and product profile.

5. Conclusions

The adoption of the OOP has produced a great demand of specific metrics. In the past, many researchers have proposed several direct and indirect metrics for effort evaluation of object-oriented systems – e.g., WMC, Size2, TJCC, HSCC, CM_{LOC}, Size2, etc. In this paper, these metrics have been compared with *new* and *more general* metrics for class complexity estimation. During the validation phase, it has also been demonstrated that the metrics proposed with optimal weights present the highest correlation with the real effort with respect to the other metrics. In addition, the same metrics have a low value for the variance – i.e., they are also quite reliable. Factors that reduce the costs of class development have been also highlighted on the basis of objective results (e.g., inheritance of attributes). Moreover, the metric proposed are capable of producing reliable evaluations considering a restricted number of terms. As a limit, the fully predictive metric, CC'_m, has been proposed. This metric is not very precise, but it can be evaluated since the early stages of software life-cycle.

As a conclusion, the validation phase reported in this paper and the one in Nesi and Campanai (1996) have demonstrated that metrics presented in this paper are enough general to be used for strongly different object-oriented languages (in fact, their main concepts were also employed for TROL (Nesi and Campanai, 1996)).

Acknowledgements

The authors would like to thank in particular the following members of TOOMS, ICOOMM, QV, LIOO and MOODS teams for collecting data about effort class per class: M. Campanai, M. Traversi of CESVIT, U. Paternostro, F. Fioravanti, S. Crucchi, M. Bruno, F. Bellini, F. Butera, N. Baldini, F. Spinu, M. Perfetti of Elexa. A special thank to B. Pages for an early version of the class browser (i.e., Xcoral), and A. Borri of CESVIT, for his help in designing the first version of TAC++. A sincere thank to the reviewers for their contribution in making the paper clearer.

References

Ambler, S.W., 1994. The pinball lifecycle model, Object Magazine 4 (6).

Bilow, S.C., Lea, D., Freburger, K., deChampeaux, D., 1993. Workshop on: processes and metrics for object-oriented development. In: Proceedings of the OOPSLA'93, Conference on Object-Oriented Programming Systems, Languages, and Applications. Washington, DC, USA.

Boehm, B.W., 1986. A spiral model of software development and enhancement. ACM SIGSOFT Software Engineering Notes 11 (4), 14–24.

Booch, G., 1994. Object-Oriented Design with Applications. Benjamin/Cummings, California, USA.

Brito e Abreu, F., Goulao, M., Esteves, R., 1995. Toward the design quality evaluation of object oriented software systems. In: Proceedings of the Fifth International Conference on Software Quality. Austin, USA.

Bucci, G., Campanai, M., Nesi, P., Traversi, M., 1993. An object-oriented case tool for reactive system specification. In: Proceedings of the Sixth International Conference on Software Engineering and Its Applications (sponsored by: EC2, CXP, CIGREF, and SEE). Le CNIT, Paris la Defense, France.

Bucci, G., Campanai, M., Nesi, P., Traversi, M., 1994. An object-oriented dual language for specifying reactive systems. In: Proceedings of the IEEE International Conference on Requirements Engineering, ICRE'94. Colorado Spring, Colorado, USA.

Campanai, M., Nesi, P., 1994. Supporting object-oriented design with metrics. In: Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS Europe'94. Versailles, France.

Cant, S.N., Jeffery, D.R., Henderson-Sellers, B., 1991. A conceptual model of cognitive complexity of elements of the programming process, Technical Report, University of New South Wales, Information Technology Research Centre, no. 57, New South Wales 2033, Australia.

Cant, S.N., Henderson-Sellers, B., Jeffery, D.R., 1994. Application of cognitive complexity metrics to object-oriented programs, Journal of Object Oriented Programming, JOOP 52–63.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20 (6), 476–493.

Coulange, B., Roan, A., 1993. Object-oriented techniques at work: facts and statistics. In: Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS, Europe 93. Versailles, France, pp. 89–94.

Halstead, H.M., 1977. Elements of Software Science. Elsevier, North Holland.

Henderson-Sellers, B., 1991. Some metrics for object-oriented software engineering. In: Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems, TOOLS 6 Pacific 1991. pp. 131–139.

Henderson-Sellers, B., 1993. The economics of reusing library classes. Journal of Object Oriented Programming 43–50.

Henderson-Sellers, B., 1994. Identifying internal and external characteristics of classes likely to be useful as structural complexity metrics. In: Proceedings of International Conference on Object Oriented Information Systems, OOIS'94. London, pp. 227–230.

Henderson-Sellers, B., Edwards, J.M., 1990. The object oriented systems life cycle. Communications of the ACM 33 (9), 143–159.

Henderson-Sellers, B., Tegarden, D., Monarchi, D., 1994. Metrics and project management support for an object-oriented software development. In: Tutorial Notes TM2, TOOLS Europe'94, International Conference on Technology of Object-Oriented Languages and Systems. Versailles, France.

Hopkins, T.P, 1994. Complexity metrics for quality assessment of object oriented design. In: Ross, M., Brebbia, C.A., Slaples, G., Slapleton, J. (Eds.), Software Quality Management II, Building Quality into Software, vol. 2. Computational Mechanisms Press, pp. 467–481.

Laranjeira, L.A., 1990. Software size estimation of object-oriented systems. IEEE Transactions on Software Engineering 16 (5), 510–522.

Li, W., Henry, S., 1993. Object-oriented metrics that predict maintainability. Journal of Systems Software 23, 111–122.

Lorenz, M., Kidd, J., 1994. Object-Oriented Software Metrics, A Practical Guide. Prentice-Hall, NJ.

McCabe, T.J., 1976. A complexity measure. IEEE Transactions on Software Engineering 2 (4), 308–320.

Meyer, B., 1990. Tools for the new culture: lessons learned from the design of the Eiffel libraries. Communications of the ACM 33 (9), 68–88.

Nesi, P., 1995. Objective software quality, Proceedings of the Objective Quality 1995, Second Symposium on Software Quality Techniques and Acquisition Criteria, Lecture Notes in Computer Science, no. 926 (editorial). Springer, Berlin.

Nesi, P., Querci, T., 1994. Measuring functional complexity, Technical Report, Dipartimento di Sistemi e Informatica, Facoltà di Ingegneria, Università di Firenze, RT 27/94, Florence, Italy.

Nesi, P., Campanai, M., 1996. Metric framework for object-oriented real-time systems specification languages. The Journal of Systems and Software 34, 43–65.

Rousseeuw, P.J., Leroy, A.M., 1987. Robust Regression and Outlier Detection. Wiley, New York, USA.

Shepperd, M., Ince, D., 1993. Derivation and Validation of Software Metrics. Clarendon Press, Oxford.

Thomas, D., Jacobson, I., 1989. Managing object-oriented software engineering. In: Tutorial Note, TOOLS'89, International Conference on Technology of Object-Oriented Languages and Systems. Paris, France, p. 52.

Williams, J.D., 1996. Managing iteration in OO projects, IEEE Computer 39–43.

Zuse, H., 1991. Software Complexity: Measures and Methods. Walter de Cruyter, Berlin.

Zuse, H., 1994. Quality measurement – validation of software metrics. In: Proceedings of the Seventh International Software Quality Week in San Francisco, QW'94. Software Research, pp. 4-T-2 .

Paolo Nesi was born in Florence, Italy, in 1959. He received his doctoral degree in electronic engineering from the University of Florence, Italy, and received the Ph.D. degree from the University of Padoa, Italy, in 1992. In 1991, he was a visitor at the IBM Almaden Research Center, CA, USA. Since November 1991, he is with the Department of Systems and Informatics of the University of Florence, Italy, as a Researcher and Assistant Professor of both "Computer Science" and "Software Engineering". Since 1995, he is Assign. Prof. of Information Technology. Since 1987, he is active on several research topics, object-oriented technology, real-time systems, quality, testing, formal languages, physical models, parallel architectures. He holds the scientific responsibility at the CESVIT for object-oriented technologies and HPCN. He is a member of the Scientific Committee of the Italian Association on Object-Oriented Technologies (TABOO). He is a Program Chair of the "2nd Euromicro Working Conference on Software Maintenance and Reengineering", CSMR'98, Florence, March 1998. He has been General Chair of Objective Quality Symposium 1995, Lecture Notes in Computer Science no. 926, Springer. He has been a member of programme committee of international Conferences (e.g., ICECCS'96 and ICECCS'97, "IEEE International Conference on Engineering of Complex Computer Systems"; AQUIS'96, "3rd International Conference on Achieving Quality in Software"; CSMR'97"). Nesi is an editorial board member of the "Journal of Real-Time Imaging", Academic Press, and responsible of several projects (e.g., DIM45 ESPRIT III MEPI, MOODS ESPRIT-HPCN, etc.). He is a member of IEEE, IAPR-IC, TABOO and AIIA.

Torello Querci was born in Florence, Italy. He received his doctoral degree in information engineering from the University of Florence, Italy, in 1995. Since 1995, he is a researcher at CQ_ware (Center for Software Quality) at CESVIT (High-Tech Agency for technology transfer). He is active on object-oriented, quality control, object-oriented methodology for analysis and design, code analysis tools and product/process assessment, in general.