

CEOS - a cost estimation method for evolutionary, object oriented software development

Siar Sarferaz and Wolfgang Hesse, Universität Marburg/Lahn, Germany

Abstract:

In this article we present a method for estimating the effort of software projects following an evolutionary, object-oriented development paradigm. Effort calculation is based on decomposing systems into manageable building blocks (components, subsystems, classes), and assessing the complexity for all their associated development cycles. Most terms of the complexity calculation formulae carry coefficients which represent their individual weights ranging from factors for particular features up to general influence factors of the project environment. These coefficients can continuously be improved by statistical regression analysis.

Outstanding features of the method are its flexibility (allowing estimations for project portions of any size) and its capability to deal with dynamic adjustments which might become necessary due to changed plans during project progress. This capability reflects the evolutionary character of software development and, in particular, implies revision, use and evaluation activities.

1 Introduction

Estimating the effort and costs for software projects has turned out to be an important competition factor in the market of individual software development - mostly due to the increasing proportion of fixed-price projects demanded by the customers. Many empirical investigations have shown that time and cost budgets are frequently exceeded (cf. e.g. [Gen 91] and [Vas 99]) mostly due to too optimistic estimations. Thus an as precise as possible cost estimation is a decisive prerequisite for economical and successful software project management.

However, due to many factors of uncertainty reliable cost estimation is a very difficult task. As an immaterial and always changeable product, software is hardly to be quantified. Estimating the development effort implies taking care of many product-specific influence factors (like product complexity and quality) as well as process-specific ones (like use of methods and tools, quality of teams and organisation), which altogether increase the difficulties.

There are several cost estimation methods some of which have been widely disseminated and experienced (as, e.g. Function Point [Alb79] or COCOMO [Boe 81]). Most of them follow the basic principle of "*Break down - sum up*": The functionality of the intended system is broken down to smaller better manageable units (e.g. called "functions"), the complexity of which is estimated and then summed up and modified by one or several factors representing general system and project characteristics. Only very few methods are suited for object oriented (OO-) software development. For example, H. Sneed with his Object Point method tries to apply the basic idea of Function Points to the "OO world" [Sne 96]. Besides its well-acknowledged benefits we see three major deficits of this method:

- (1) Like the Function Point method it relies on a variety of subjective assessments which are weighted by numerical coefficients given by the method in a sometimes arbitrary and doubtful way.
- (2) Some of its calculation methods (as e.g. the treatment of quality influence factors) are hardly to understand and would require some revisions to be used in practice.

- (3) There are no provisions for dynamic assessments and estimation adaption according to the growing knowledge and possible changes during project progress.

Traditional cost estimation methods were based on the then popular paradigms of functional decomposition and phase-driven development. Object-oriented methods have emphasised the ideas of data-based decomposition, reuse and component-driven development. Thus we have based our method on a process model (called EOS, cf. [Hes 96], [Hes 97a,b]) which follows the basic principles of *object orientation through all stages of development, hierarchical system architecture, component-based development cycles, software evolution through use and revision* and which emphasises a *recursive and orthogonal process structure*. This structure offers high flexibility for modular system development, use and reuse of already existing components as well as for project management but it requires rather sophisticated planning, coordinating and estimating procedures and instruments.

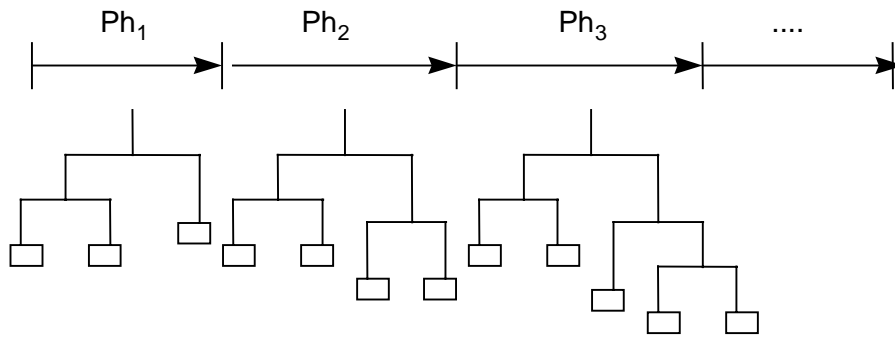
In the following sections we start with a short summary of our process modelling approach (section 2) and then present the CEOS method in three steps corresponding to the EOS levels of software development: class level, component level, system level (section 3). In the fourth section implementation issues are addressed and some conclusions for further work are drawn.

2 EOS - a component-based approach to software process modelling

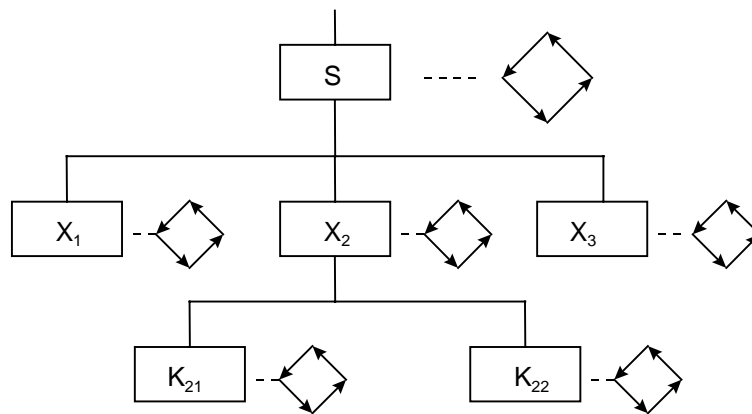
In contrast to most traditional, waterfall-like software life cycle models (and even many process models for OO development as, for example, Rational's Unified Process [JBR 99]), the EOS model does not emphasise a phase-driven but a component-oriented development approach (cf. fig. 1). Among others, this approach has the following implications:

- The principal criterion for structuring software development processes is no longer their phases (cf. the upper part of fig. 1) but a *hierarchy of system building blocks* (in EOS called *components, classes and subsystems*) representing a static view on the system architecture (cf. the lower part of fig. 1).
- Components and subsystems are the central structural units of the system architecture which are not only used to group smaller units (e.g. classes) to larger logical units but also for organisational reasons as e.g. delegation of a component to a particular person or to a small team responsible for its development and maintenance, planning of associated activities or storage and support for retrieval in a component library. Subsystems are (normally non-disjoint) collections of classes grouped together for joint execution during test and system integration.
- Each building block has its own *development cycle* consisting of four main activities called *analysis, design, implementation* and *operational use* (cf. fig. 2).
- In contrast to traditional waterfall models, new building blocks may be formed - and corresponding development cycles are enacted - at any stage of the development. A new development cycle may interrupt an already existing one or it is evolving in parallel. Concurrent development cycles are coordinated by project management with the help of *revision points*. Thus the traditional phase structure dominating one overall, system-wide development process is replaced by a collection of concurrent, individual development cycles for all building blocks under construction.

For further details cf. [Hes 96], [Hes 97a/b].



Phase oriented vs ...



... Component oriented process structure

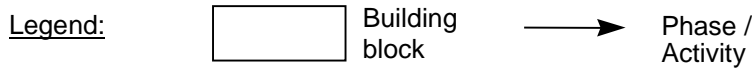


Fig. 1: Two approaches to structure the software process

According to the hierarchy of software building blocks, three levels of granularity can be distinguished which are also important for the following cost estimation procedures: (a) *the system level*, (b) *the component level* and (c) *the class level*.

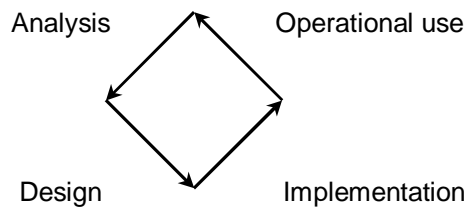


Fig. 2: Structure of an EOS development cycle

3 The cost estimation model: General assumptions

In this section we present our CEOS (Cost estimation for EOS project) method. It is based on a proposal of P. Nesi and T. Querci for estimating the development effort of object oriented systems [N-Q 98]. Some basic assumptions of this approach are:

- The effort needed for developing a piece of software depends on its complexity in a linear way.
- The complexity of a software building block depends on the complexity of its *ingredients* (e.g. smaller, contained building blocks or other structural units) which can be weighted by individual *coefficients*.
- The quality of an estimate depends on the fitness and preciseness of the used coefficients. These can continuously be improved by using "real life" data from earlier projects and statistical regression analysis techniques [R-A 99]. In a running project, data gathered from terminated phases or activities can be used to improve the current estimates.
- Particular coefficients can be defined and used for tailoring estimates to certain segments or iterations of a development cycle.
- For detailed calculations, a basic metric (denoted „m” in the sequel - for quantifying the program complexity) is required. This can e.g. be the well-known *LOC (Lines Of Code) metric* or the *McCabe metric*. In the sequel we use a variant of the McCabe metric:

$$MCC = |E| - |N| + 1,$$

where $|N|$ is the number of nodes and $|E|$ is the number of edges in a flow diagram corresponding to the control structure of the program.

We distinguish metrics for the *definition complexity* and the *application complexity* of classes and their attributes. For example, the application complexity is used in the metrics for inheritance, association and aggregation relationships. The (definition) complexity of a class to be developed is calculated from the complexity of its attributes and operations which may involve the application complexity of other (used) classes. The application complexity value is always less than the corresponding definition complexity value.

In order to facilitate the presentation, we start with two preliminary assumptions:

- (1) For each building block, the corresponding development cycle is executed *exactly once*.
- (2) We are interested in *total estimates* covering the effort for all activities of a development cycle.

These assumptions will be leveraged in later sections.

4 Metrics for the sub-class layer

The following table (fig. 3) lists the metrics which are introduced in this section.

Metric	Name	Metric	Name
Attributes		Operations	
AC_m^{App}	Attribute Application Complexity	MC_m^{App}	Method Application Complexity
AC_m^{Def}	Attribute Definition Complexity	MC_m^{Def}	Method Definition Complexity
		MBC_m^{App}	Method Body Complexity (Application)
		MBC_m^{Def}	Method Body Complexity (Definition)
		MIC_m^{App}	Method Interface Complexity (Application)
		MIC_m^{Def}	Method Interface Complexity (Definition)
		NMLA	Number of Method Local Attributes

Fig. 3: Metrics for the sub-class layer

4.1 Attribute metrics

We define know the metrics AC_m^{App} and AC_m^{Def} to measure the complexity of attributes. The letter „m” stands for a basic metric, the abbreviations „App” and „Def” refer to application and definition complexity, respectively.

Let $A : ClassA$ be an attribute of type „ClassA”.

Definition I (Attribute Application Complexity): $AC_m^{App} = K$, where $K \in \mathbb{R}^+$ is a constant.

Definition II (Attribute Definition Complexity): $AC_m^{Def} = CC_m^{App} (ClassA)$

CC_m^{App} is a metric for measuring the class definition complexity, which is defined below. Note that in the definition of CC_m^{App} the metric AC_m^{App} is used thus avoiding any circular definition.

4.2 Operation metrics

Let $F (A_1:C_1, \dots, A_n:C_n) : C_{n+1}$ be a method interface, where F is a method identifier, A_i are attribute identifiers and C_i are class identifiers. Further let A_{n+1} be an (anonymous) attribute of type C_{n+1} , which stands for the return value of the method.

Definition I (*Method Application Complexity*): $MC_m^{App} = MIC_m^{App} + MBC_m^{App}$, with

$$MIC_m^{App} = \sum_{i=1}^{n+1} AC_m^{App} (A_i) \quad \text{and} \quad MBC_m^{App} = \sum_{i=1}^{NMLA} AC_m^{App} (A_i) + m$$

NMLA stands for the number of locally defined attributes of a method.

Definition II (*Method Definition Complexity*): $MC_m^{Def} = w_{MIC_m} MIC_m^{Def} + w_{MBC_m} MBC_m^{Def}$

The metrics MIC_m^{Def} und MBC_m^{Def} can be defined analogously to MIC_m^{App} and MBC_m^{App} by replacing AC_m^{App} with AC_m^{Def} .

w_{MIC_m} and w_{MBC_m} are coefficients to be determined by the statistical technique of robust regression (cf. below).

5 Metrics for the class layer

Each class is subject to a development cycle consisting of the activities analysis, design, implementation and operational use. According to the project progress along these activities more data become available which can be used for refined effort calculations. Therefore different metrics apply - corresponding to the time of estimation. Note that all given metrics refer to the *total* effort for the class development.

Metric	Name	Metric	Name
Class layer			
CC_m^{App}	Class Application Complexity	$InCC_m$	Inherited Class Complexity (implementation)
CC_m^{Def}	Class Definition Complexity	LC_m	Local Class Complexity (analysis)
CCA_m^{App}	Class Application Complexity during Analysis activity	LCC_m	Local Class Complexity (design)
CCA_m^{Def}	Class Definition Complexity during Analysis activity	$LoCC_m$	Local Class Complexity (implementation)
CCD_m^{App}	Class Application Complexity during Design activity	NDP	Number of Direct Parents
CCD_m^{Def}	Class Definition Complexity during Design activity	NIC	Number of Inherited Classes
CCI_m^{App}	Class Application Complexity during Implementation activity	NLA	Number of Local Attributes
CCI_m^{Def}	Class Definition Complexity during Implementation activity	NLM	Number of Local Methods
CU_m	Class Usability	$NPuCA$	Number of Public Class Attributes
IC_m	Inherited Class Complexity (analysis)	$NPuCM$	Number of Public Class Methods
ICC_m	Inherited Class Complexity (design)		

Fig. 4: Metrics for the class layer

5.1 Analysis activity

Definition I (Class *Application* Complexity during *Analysis* activity):

$$CCA_m^{App} = NLA + NLM \text{ (number of local attributes and methods)}$$

Definition II (Class *Definition* Complexity during *Analysis* activity):

$$CCA_m^{Def} = LC_m + IC_m, \text{ with}$$

$$LC_m = w_{NLA_m} NLA + w_{NLM_m} NLM \quad \text{and} \quad IC_m = w_{NIC_m} NIC.$$

w_{NLA_m} , w_{NLM_m} and w_{NIC_m} are coefficients for weighting the local and inherited elements, respectively.

5.2 Design activity

Let A_i , M_i and C_i be the i -th attribute, method or class, resp.. The complexity of a class depends on the complexity of its features, i.e. of its attributes and operation interfaces. For class definitions, the complexity of local features (LCC) and of inherited features (ICC) are considered.

Definition I (Class *Application* Complexity during *Design* activity):

$$CCD_m^{App} = \sum_{i=1}^{NLA} AC_m^{App}(A_i) + \sum_{i=1}^{NLM} MIC_m^{App}(M_i)$$

Definition II (Class *Definition* Complexity during *Design* activity):

$$CCD_m^{Def} = LCC_m + ICC_m, \text{ with}$$

$$LCC_m = w_{LAC_m} \sum_{i=1}^{NLA} AC_m^{Def}(A_i) + w_{LMC_m} \sum_{i=1}^{NLM} MIC_m^{Def}(M_i)$$

$$ICC_m = w_{ICC_m} \sum_{i=1}^{NDP} CCD_m^{App}(C_i) + NIC(C_i)$$

5.3 Implementation activity

Let A_i , M_i and C_i be the i -th attribute, method or class, resp.. Again, the complexity of a class is derived from the complexity of its features, but now the implementation-specific values and coefficients are taken.

Definition I (Class *Application* Complexity during *Implementation* activity):

$$CCI_m^{App} = \sum_{i=1}^{NLA} AC_m^{App}(A_i) + \sum_{i=1}^{NLM} MC_m^{App}(M_i)$$

Definition II (Class *Definition* Complexity during *Implementation* activity):

$CCI_m^{Def} = LoCC_m + InCC_m$, with

$$LoCC_m = w_{LoACm} \sum_{i=1}^{NLA} AC_m^{Def}(A_i) + w_{LoMCm} \sum_{i=1}^{NLM} MC_m^{Def}(M_i)$$

$$InCC_m = w_{InCCm} \sum_{i=1}^{NDP} CCI_m^{App}(C_i) + NIC(C_i)$$

5.4 Operational use activity

Normally, no new calculations are done during operational use. The results of the estimations should be assessed (i.e. compared to the actual values) and documented. The coefficients should be validated and adjusted, if necessary. In case of a repeated development cycle, a new estimation of the next cycle is required (see below).

5.5 Metrics for Reuse

Reuse of classes is not for free but requires some effort, e.g. for understanding and adapting the reused code and documentation. Therefore we define a metric for class usability.

Definition (*Class Usability*): Let A_i and M_i be the i -th attribute / method of a class.

$$CU_m = w_{CU} \left(\sum_{i=1}^{NPUCA} AC_m^{App}(A_i) + \sum_{i=1}^{NPUCM} MC_m^{App}(M_i) \right)$$

With this definition, we can now give a general definition for the complexity of a class CC_m^{Def} (CC_m^{App} can analogously be defined by replacing „Def“ with „App“).

$$CC_m^{Def}(C) = \begin{cases} \text{constant,} & \text{if } C \text{ is a system-classe} \\ CCA_m^{Def}(C), & \text{if } C \text{ is analysed} \\ CCD_m^{Def}(C), & \text{if } C \text{ is designed} \\ CCI_m^{Def}(C), & \text{if } C \text{ is implemented} \\ CU_m(C), & \text{if } C \text{ is reused} \end{cases}$$

If class C is a specialisation of class S , S plays the role of a reused class and thus $CU_m(S)$ will be assigned to $IC_m(C)$ as the inheritance complexity of C .

5.6 Example: Application of the metric CCD_m^{Def}

Now we illustrate the application of the CCD_m^{Def} (Class Definition Complexity during Design activity) metric by a brief example.

During the class design activity the static structure of classes and their relationships is modelled and described by class diagrams. This information can be used to determinate the complexity of classes. As an example, we consider a simplified class diagram for a system managing bank accounts (fig. 5).

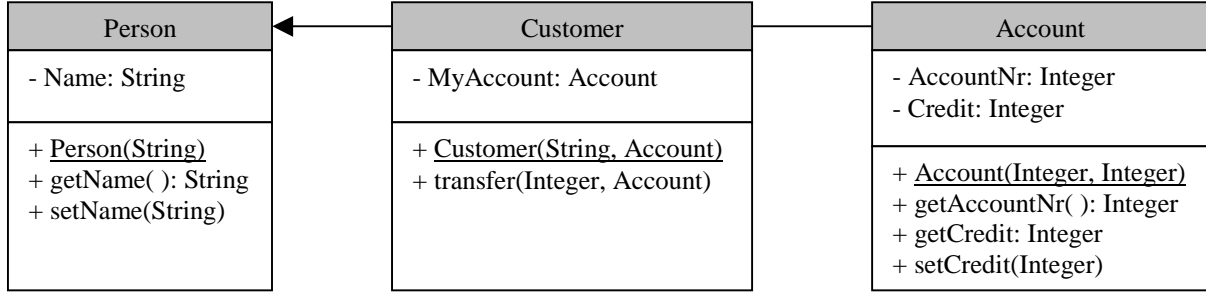


Fig. 5: A simple class diagram

Now we apply the CCD_m^{Def} metric on the class „Customer”. We assign to elementary types (like „Integer” or „String”) the complexity value $K = 1$.

$$CCD_m^{Def}(\text{Customer}) = LCC_m(\text{Customer}) + ICC_m(\text{Customer})$$

$$LCC_m(\text{Customer})$$

$$= w_{LAC_m} AC_m^{Def}(\text{MyAccount}) + w_{LMC_m} (MIC_m^{Def}(\text{Customer}) + MIC_m^{Def}(\text{transfer}))$$

$$= w_{LAC_m} CC_m^{App}(\text{Account})$$

$$+ w_{LMC_m} (AC_m^{Def}(\text{String}) + AC_m^{Def}(\text{Account}) + AC_m^{Def}(\text{Integer}) + AC_m^{Def}(\text{Account}))$$

$$= w_{LAC_m} * 7 + w_{LMC} * (1 + 7 + 1 + 7) = w_{LAC_m} * 7 + w_{LMC} * 16$$

Here, for example, $AC_m^{Def}(\text{MyAccount}) = CC_m^{App}(\text{Account}) = 7$ is calculated by counting the two attributes and the 5 parameters of the methods - all being of elementary type.

$$ICC_m(\text{Customer})$$

$$= w_{ICC_m} (CCD_m^{App}(\text{Person}) + NIC(\text{Person}))$$

$$= w_{ICC_m} (AC_m^{App}(\text{Name}) + MIC_m^{App}(\text{Person}) + MIC_m^{App}(\text{getName}) + MIC_m^{App}(\text{setName}) + 0)$$

$$= w_{ICC_m} (CC_m^{App}(\text{String}) + AC_m^{App}(\text{String}) + AC_m^{App}(\text{String})) + AC_m^{App}(\text{String})$$

$$= w_{ICC_m} (1 + 1 + 1 + 1) = w_{ICC_m} * 4$$

$$CCD_m^{Def}(\text{Customer}) = w_{LAC_m} * 7 + w_{LMC_m} * 16 + w_{ICC_m} * 4$$

6 Metrics for the component and system layer

The following table (fig. 6) summarises the metrics defined in this section.

Metric	Name	Metric	Name
Component layer		System layer	
NXC	Number of Component Classes	NSX	Number of System Components
XC_m	Component Complexity	SCA_m	System Complexity during Analysis phase
XCA_m	Component Complexity during Analysis activity	SCA_r	System Complexity coarse Analysis
XCA_r	Component Complexity coarse Analysis	SCD_m	System Complexity during Design phase
XCD_m	Component Complexity during Design activity	SCI_m	System Complexity during Implementation phase
XCI_m	Component Complexity during Implementation activity		
XU_m	Component Usability		

Fig.6 : Metrics for the Component and System layer

6.1 Component metrics

One of the characteristics of the EOS method is its component-based process architecture. Components are collections of classes and as such have development cycles analogous to those of classes. Since the corresponding metrics have a similar structure, we restrict ourselves to compact definitions.

Let C_i be the i -th class of a component X

Definition (Component Complexity during *Analysis / Design / Implementation* activity):

$$XCY_m = w_{XYm} \sum_{i=1}^{NXC} CCY_m^{Def} (C_i), \text{ where } Y = A(\text{analysis}), D(\text{esign}) \text{ or } I(\text{mplementation}).$$

Again, particular coefficients w_{XYm} are used for refinement and correction purposes on this level.

If (e.g. in the early stages of a project) only the rough number NXC of classes contained in a component is known, a coarse estimation can be performed using the metric XCA_r .

Definition (Component Complexity *coarse Analysis*): $XCA_r = w_X NXC$

For the operational use of components, similar considerations apply as in the class case.

A reused component is a collection of reused classes - which leads us the following metric:

Definition (Component Usability): $XU_m(X) = w_{XU} \sum_{i=1}^{NXC} CU_m(C_i)$

6.2 System metrics

In a similar way, a system is viewed as a collection of components and its metrics is derived from their ones. Again we give a compact definition which is to be specialised according to the activities of a system development cycle.

Let X_i be the i -th component of a system S .

Definition (System Complexity during *Analysis / Design / Implementation* phase):

$$SCY_m = w_{SYm} w_{Quality} \sum_{i=1}^{NSX} XCY_m(X_i), \text{ where } Y = A(\text{nalysis}), D(\text{esign}) \text{ or } I(\text{mplementation}).$$

A first coarse estimation can be performed with the metric SCA_r .

Definition: (*System Complexity coarse Analysis*): $SCA_r = w_S w_{Quality} NSX$

In addition to the coefficients w_{SYm} and w_S , we consider a quality factor $w_{Quality}$ on this layer. This factor can, for example, be calculated with the help of a quality characteristics table similar to the one adopted by H. Sneed from [ISO 9126] for his Object Point method [Sne 96].

During the operational use of a system, no refinements of the calculations apply but estimation results should be assessed and documented, coefficients should be validated and adjusted, and the effort for a next cycle is to be estimated, if required.

6.3 Determining the coefficients

In order to keep the CEOS method flexible, we weighted the metrics with coefficients. Different coefficients may be used to reflect the particular work conditions of different companies and institutions. Coefficients are to be determined from earlier projects using the statistical technique of regression analysis. As real data frequently contain outliers, traditional methods, like Gauss's least square technique, would easily distort the results. To avoid this problem, new statistical techniques - called robust techniques - have been developed which provide quite good results even if a certain amount of data is unreliable. For CEOS we chose the LMS (Least Median of Squares) method - one of the best possible methods from a theoretical point of view. In the following we present only the *estimator* for the LMS method.

Least Median of Squares (LMS)

Let n be the number of given data elements and ε_i represent their deviations from corresponding values f_i of a approximation function f . Then f has to be chosen in such a way that the Median of Squares

$$\text{med}_{i=1, \dots, n} \varepsilon_i^2$$

adopts a minimum .

Fig. 8: LMS estimator

7 Extending the model to multiple cycles and partial estimates

The metrics introduced so far were based on the assumptions of (a) unique and (b) always complete development cycles (cf. end of section 3). Both restrictions can now be dropped in a straightforward way - at least if the required statistical data are available. If an estimation for an incomplete cycle is required (e.g. in case of calculating the rest effort for an already started cycle) particular coefficients may be used which cast the total effort to a particular activity. Such coefficients can be determined from statistical data on the proportion of effort required for each particular activity. Since the size and hierarchy level of a building block essentially influence this proportion, statistical data should be gathered separately for the class, component and systems layers.

If p development cycles are planned for a particular building block (with $p > 1$), this can be used for calculating proportional efforts for each particular cycle. The simplest way would be to assume a constant factor $1/p$ for each cycle, but other more sophisticated factors might be derived from statistical data collected during prior projects.

8 Prototype Implementation and conclusions

In the preceding sections, we have presented an effort calculation and cost estimation method which is based on a component-oriented process model and architecture. A simple prototype has been implemented which explores the user interface and demonstrates some basic functions of the CEOS method. A complete implementation which supports all the presented calculations and which implies the use of statistical techniques will be the next major step in the ongoing CEOS project.

We are aware that applying the CEOS method is not easy since it starts from a rather sophisticated process model and - at least if one expects precise results - it requires encompassing data collections and rather complex calculations. But on the other hand it offers the following advantages:

- It considers development cycles (instead of simple phases) and thus addresses modern software development paradigms like software evolution or component-based development.
- Cost estimations can be done for complete projects or for any of their parts. Parts are the building blocks defined according to the software architecture in form of components, classes or subsystems. Partial estimations can be done and used for time, budget or personnel planning on any level of detail.
- For any building block, the total effort for one development cycle or the partial effort for any particular activity can be calculated as well as the effort for several development cycles concerning the same building block.
- The CEOS method supports particular object-oriented techniques like inheritance or reuse of classes and components.
- Most calculation formulae contain coefficients which allow to tailor all estimations to the particular conditions and requirements of the institution concerned. Coefficients can be adapted to specific influence factors like quality requirements, personnel qualification, customer familiarity or tool support. Continuous adaptation of the coefficients according

to the most recent available statistical data make the method a valuable tool for dynamic project management and evolutionary software development.

In a current Ph.D. project, the CEOS method is enhanced and further implemented with the aim to extend an existing UML-oriented development tool („objectiF” of microTOOL GmbH) by a powerful management tool to be used in software practice.

References:

- [Alb 79] A. J. Albrecht: *Measuring Applications Development Productivity*, in: Proceedings of IBM Applications Division Joint SHARE/GUIDE Symposium, S. 83 – S. 92, Monterey, CA, 1979.
- [Boe 81] B. W. Boehm: *Software Engineering Economics*, Prentice Hall, 1981.
- [Dum 99] R. Dumke, Alain Abran: *Software Measurement / Current Trends in Research an Practice*, Wiesbaden: Deutscher Universitäts Verlag, 1999.
- [Gen 91] M. van Genuchten: *Why is Software Late? An Empirical Study of Reasons For Delay in Software Development*, in: IEEE Transaction on Software Engineering, Vol. 17, No. 6, S. 582 – S. 590, June 1991.
- [Hes 97a] W. Hesse: *Life cycle models of object-oriented software development methodologies*, Marburg: A. Zender et al.: Advanced concepts, life cycle models and tools for object-oriented software development, Reihe Softwaretechnik 7, Tectum Verlag Marburg, 1997.
- [Hes 97b] W. Hesse: *Improving the software process guided by the EOS model*, in: Proc. SPI '97 European Conference on Software Process Improvement, Barcelona 1997.
- [Hes 97c] W. Hesse: *Wie evolutionär sind die objekt-orientierten Analysemethoden? Ein kritischer Vergleich*, in: Informatik-Spektrum 20.1, S. 21 – S. 28, 1997.
- [ISO 9126] ISO/IEC 9126: *Information technology – Software Product Evaluation – Quality Characteristics and Guidelines for Their Use*, 1991.
- [N-Q 98] P. Nesi, T. Querci: *Effort estimation and prediction of object-oriented systems*, in: The Journal of Systems and Software 42, S. 89 – S. 102, 1998.
- [R-A 99] P. J. Rousseeuw, Stefan Van Aelst: *Positive-Breakdown Robust Methods in Computer Vision*, Department of Mathematics and Computer Science, U.I.A., Belgium, 1999.
- [Sne 96] H. M. Sneed: *Schätzung der Entwicklungskosten von objektorientierter Software*, in: Informatik Spektrum 19, S. 133 – 140, 1996
- [Vas 99] H. Vaske: *Systemhäuser zeigen Schwäche im Support*, in: Computerwoche, Nr. 11, S. 1, S. 9 – S. 10, März 1999.
- [Zus 98] H. Zuse: *A Framework of Software Measurement*, Berlin/New York: Walter de Gruyter, 1998.

Authors:

Dipl.– Inf. Siar Sarferaz
microTOOL GmbH,
Voltastr. 5,
D–13349 Berlin

Tel.: +49-030-467086-0, Email: Siar.Sarferaz@microTOOL.de

Prof. Dr. Wolfgang Hesse,
FB Mathematik/Informatik, Universität Marburg,
Hans Meerwein-Str.,
D-35032 Marburg

Tel.: +49-6421-282 1515, Fax: +49-6421-282 5419,
email: hesse@informatik.uni-marburg.de