Software Size Estimation of Object-Oriented Systems

LUIZ A. LARANJEIRA, STUDENT MEMBER, IEEE

***Abstract*—Software size estimation has been the object of a lot of research in the software engineering community due to the need of reliable size estimates in the utilization of existing software project cost estimation models. This paper discusses the strengths and weaknesses of existing size estimation techniques, considers the nature of software size estimation, and presents a software size estimation model which has the potential for providing more accurate size estimates than existing methods. The proposed method takes advantage of a characteristic of object-oriented systems, the natural correspondence between specification and implementation, in order to enable users to come up with better size estimates at early stages of the software development cycle. Through a statistical approach the method also provides a confidence interval for the derived size estimates. The relation between the presented software sizing model and project cost estimation has also been considered.**

***Index Terms*—Functional specification, object-oriented systems, software cost estimation, software size estimation.**

I. Introduction

THE software crisis has focused the attention of the software engineering community on the research of disciplined techniques for software development in the attempt to reduce and control the alarming growth of software systems development and maintenance costs. In particular, the commonly noticed tendency in software systems development for gross cost overruns and undesirable project delays has caused a lot of work to be done in developing project cost and effort estimation models. Accurate cost and scheduling estimations provide highly valuable aid in a number of management decisions, budget and personnel allocations, and in supporting reliable bids for contract competition.

Cost estimation models today available for the software engineering practitioner include: the Walston–Felix model [36], the Doty model [11], the Putnam model [25], the RCA PRICE S model [8], the Bailey–Basili model [2], the COCOMO (COnstructive COst MOdel) model [3], the SOFTCOST model [34], and, recently, the Jensen model [14]. An overview of each of these models is presented in [7]. In [3], a complete description of the COCOMO model together with a very detailed approach for its utilization on the daily life of an organization can be found. The discussion we carry out here will be generally valid for any of the mentioned cost models, although we concentrate something more on COCOMO in terms of examples and details due to a more complete documentation available for this model.

Manuscript received July 25, 1988; revised October 15, 1989. Recommended by C. G. Davis.

The author is with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712.

IEEE Log Number 9034386.

The reception of cost estimation models in the software community has been usually good. Managers feel more comfortable using estimation models than just relying on rules of thumb and entirely subjective judgments when planning budgetary and personnel resources for a new project. Even though estimation models have some limitations that managers need to be aware of [3], [24], [7], they may be viewed as valuable tools in the software engineering process.

A common point concerning the above mentioned models is that they base their effort and scheduling predictions on the estimated size of the software project at hand, in terms of number of lines of code (LOC), or thousands of lines of code (KLOC). Generally the effort estimation is based upon an equation similar to

$$E = A + B*(\mathrm{KLOC})^C$$

where E stands for estimated effort (usually in man-months), A, B, and C are constants, and KLOC is the expected number of thousands of lines of code in the final system. From the above equation it is easy to see that a given percentage error in the size (KLOC) may cause an even larger percentage error in the estimated effort. For instance, in COCOMO a 50% error in the size estimate will roughly result in a 63% error in the effort estimate. In other terms, size estimation error causes cost estimation error, since cost estimates are derived based on effort prediction.

Existing software cost estimation models assume that a plans and requirement analysis phase (Fig. 1), and consequently the system specifications, has been carried out before the model is applied. Their cost and schedule estimations cover the subsequent phases of the software development cycle (product design, detailed design, coding, integration, and test). Maintenance costs are estimated in a slightly different way, since by maintenance time the system is already entirely developed and its actual size is known.

It is obvious that, concerning development cost estimation, if a model would only produce effort estimates, let us say, after the detailed design or coding phase, such an estimation would have relatively low value for the project management and control. We surely need estimates in the early stages of the software life cycle in order that these estimates may cause a real impact on the development process. As a consequence, size estimations must be provided in the early phases of the software life cycle.

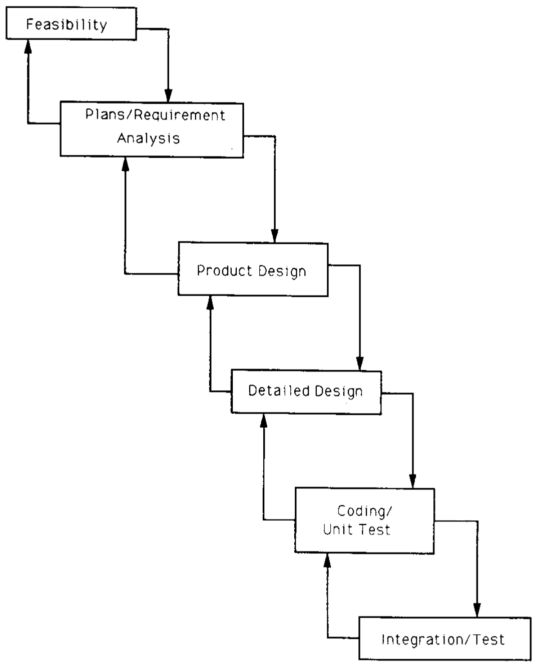

Fig. 1. Software life cycle phases.

Despite the above stated need, existing cost estimation models do not provide an integrated, detailed rationale and guidelines for producing system size estimates with required accuracy. This inconsistency has been pointed out by Bryant and Kirkham [6].

This paper attempts to focus on the problem of early size estimation. We will consider previously proposed size estimation techniques, discuss the nature of software size estimation and present a technique, based on an object-oriented specification model (for an object-oriented software implementation) and on statistical methods, that might turn out to provide more reliable software size estimates.

II. Existing Software Size Estimation Techniques

Several approaches for predicting software size have been proposed in the literature. They could be divided in two subsets: subjective techniques and objective models.

A. Subjective Techniques

The most popular sizing technique used is the PERT method where practitioners rely on expert judgment to estimate the ultimate size of a project. Such estimates are based on analogy with projects of similar characteristics, experience, or, when all else fails, on the intuition of the estimators [24].

In this approach the project is decomposed in its major functions, during the plans and requirements phase of the software life cycle (Fig. 1), and an estimation is made for each of them. The estimated size of the total system will be the sum of the estimates of the composing functions [39].

As an example, let us consider a software system to be developed for secure message communication. Project scope indicates a microprocessor based system linked in a network, providing for communication between government agencies. Evaluation of the system indicates the following major functions: user interface and control facilities, edition module, encryption/decryption modules, and communication modules. Table I shows the size estimates for each major function and for the entire system.

Putnam and Fizsimmons [26] suggested a method for adding some objectiveness to this technique. The expected number of lines of code (S_i) for each function is calculated as a weighted average of the optimistic (O_i), the most likely (M_i), and the pessimistic (P_i) size estimates. The estimated size of the total system (S_s) will again be the sum of the estimated function sizes. For example

$$S_i = \frac{O_i + 4M_i + P_i}{6} \qquad S_s = \sum S_i.$$

Considering that there is a very small probability that the actual size values do not fall in between the optimistic and pessimistic estimates, the deviation for each function estimate (D_i) and for the total system size estimate (D_s) will be

$$D_i = \frac{O_i - P_i}{6} \qquad D_s = (\sum D_i^2)^{1/2}.$$

These equations are based on a beta distribution which means that 68% of the time the real size of the final software should be within $S_s + D_s$ and that the estimates are unbiased toward either underestimation and overestimation. Table II extends the simple expert judgment example using this PERT approach.

Despite the common utilization of subjective techniques for early software size estimation, two points can be clearly observed which expose the weaknesses of such an approach:

1) Experience has shown that expert judgment varies widely due to psychological and personal factors, and usually cannot provide estimates with required accuracy. Reference [7] quotes a study performed by Yourdon Inc., where several experienced managers estimated the size of 16 software projects on the basis of the complete specification for each project. The average expert predictions are presented in Table III together with the actual size of each software project, after completed. It may be easily seen that the discrepancy between estimated and real size values is large enough to raise questions about the applicability of the method.

2) The assumption, considered in the PERT technique, that estimates are unbiased toward underestimation or overestimation, is not confirmed by current experience. It can be noticed in Table III that 12 of the 16 projects were underestimated by experts. The reasons for this underestimation tendency include the desire to please management, incomplete recall of previous experiences, lack of familiarity with the entire software job, and lack of enough

TABLE I
SIMPLE EXPERT ESTIMATION FOR SECURE MESSAGE COMMUNICATION SYSTEM SIZE

Function Module	Estimated Size
User Interface/Control	2,500
Editor	1,500
Encription/Decription	800
Communications	1,000
Total Project	5,800

TABLE II
PERT ESTIMATION FOR SECURE MESSAGE COMMUNICATION SYSTEM SIZE

Function Module	Optimistic Size	Most Likely Size	Pessimistic Size	Expected Size	Deviation
UI/Cont	1,800	2,500	3,200	2,550	283
Editor	1,000	1,500	2,000	1,500	167
Enc/Dec	500	800	1,000	784	83
Commun	700	1,000	1,300	1,000	100
Tot Proj	4,000	5,800	7,800	5,584	353

TABLE III
ACTUAL AND PREDICTED SIZE OF 16 SOFTWARE PROJECTS

Product	Actual Size	Predicted Size
1	70,919	34,705
2	128,837	32,100
3	23,015	22,000
4	34,560	9,100
5	23,000	12,000
6	25,000	7,300
7	52,080	28,500
8	7,650	8,000
9	25,860	30,600
10	16,300	2,720
11	17,410	15,300
12	33,900	105,300
13	57,194	18,500
14	21,020	35,400
15	8,642	3,650
16	17,480	2,950

knowledge of the particular project being estimated. These considerations show the need of more accurate software sizing techniques.

B. Objective Models

There have been several attempts to come up with an objective model for predicting software size. Generally the size of a system is expressed as a function of some known quantities that reflect characteristics of the system. In [18], Levitin classifies these models in two groups, according to the type of the quantities upon which the estimates are made. These groups can be called external (specification level) models or internal (implementation level) models.

Specification Level Objective Models: Specification level models express size as a function of a number of quantities which can usually be determined early in the software life cycle from system specifications. One of these methods, derived by Albrecht (see [1] and [33]), base software sizing on the so-called "function points." These are characteristics of the system such as the number of input and output files, the number of logical internal files, the number of inquiries, etc. The calculation of the "function points" is performed as a weighted sum of these quantities, adjusted by some complexity factors. The estimated size is related to the number of "function points" (FP) by an equation of the type

$$\text{ES} = (B * FP) - A$$

where ES is the estimated software size in thousands of lines of code (KLOC), and A and B are constants. Table IV shows briefly the steps for calculating the "function points" for a given application. First the total unadjusted function points (TUFP) are calculated. The calculation of TUFP is based on weighting the system components (external inputs, external outputs, etc.) and classifying them as "simple," "average," or "complex" depending on the number of data elements in the component and other factors [Table IV(a)]. Then the technical complexity factor (TCF) is calculated by estimating the degree of influence (DI) of some 14 characteristics of the component [Tables IV(b) and IV(c)]. Finally, the function points (FP) are given by multiplying the total unadjusted function points (TUFP) by the technical complexity factor (TCF) [see formula below Table IV(c)].

Another method, similar to Albrecht's one, was derived by Itakura and Takayanagi [12]. This model bases its estimations on quantities such as the number of input and output files, the number of input and output items, the number of items of transaction type reports, etc. These quantities are presented in Table V with corresponding designators (X_i's). The model equation is as follows

$$\text{ES} = A_i * X_i$$

where ES is the estimated software size in thousands of lines of code (KLOC), the X_i's are the quantities based on which the estimation is made, and the A_i's are the corresponding coefficients.

Although these models attempted to achieve a somewhat more scientific approach to software sizing, it can be generally stated that they still present a number of difficulties which point to the need of further research in the field. We could summarize these points in the following considerations:

1) These methods were derived to be used in banking and business applications. Therefore, they lack generality concerning the nature of the systems to which they might be applicable.

2) Generally, the quantities upon which the estimates are based are related to the interactions between the system and the environment (inputs and outputs). Albrecht also attempted to consider the effects of complexity factors in his sizing model. It is not difficult to understand that programs having the same type and number of interactions with the environment might have totally different sizes. It has also been noticed by Boehm, in [3], that complexity does not necessarily relate to size. In this sense we might have very complex functions which have a relatively short sized implementation, or relatively lengthy simple housekeeping functions.

TABLE IV
FUNCTION POINTS CALCULATION: (a) UNADJUSTED FUNCTION POINTS CALCULATION, (b) TECHNICAL COMPLEXITY FACTOR CALCULATION, (c) VALUES CORRESPONDING TO DIFFERENT DEGREES OF INFLUENCE.

Description	Level of Information Processing Function			
	Simple	Average	Complex	Total
External Input	--- *3 = ---	--- *4 = ---	--- *6 = ---	---
External Output	--- *4 = ---	--- *5 = ---	--- *7 = ---	---
Logical Internal File	--- *7 = ---	--- *10= ---	---*15= ---	---
External Interface File	--- *5 = ---	--- *7 = ---	--- *10= ---	---
External Inquire	--- *3 = ---	--- *4 = ---	--- *6 = ---	---
	Total Unadjusted Function Points (TUFP)			---

(a)

ID	Characteristic	DI	ID	Characteristic	DI
C1	Data Communications	--	C8	On-Line Update	--
C2	Distributed Functions	--	C9	Complex Processing	--
C3	Performance	--	C10	Reusability	--
C4	Heavily Used Configuration	--	C11	Installation Ease	--
C5	Transaction Rate	--	C12	Operational Ease	--
C6	On-Line Data Entry	--	C13	Multiple Sites	--
C7	End User Efficiency	--	C14	Facilitate Change	--
			Total Degree of Influence (TDI)		--

TCF = 0.65 + 0.01 * TDI

(b)

DI Values	
No Influence = 0	Average Influence = 3
Insignificant Influence = 1	Significant Influence = 4
Moderate Influence = 2	Strong Influence = 5

FP = TUFP * TCF

(c)

3) Another difficulty arises when we think about how to extend these models to applications such as scientific programs, text processing, communications systems, and others, which emerge day to day in the computer software field. There is no easy way to find the quantities upon which we should base our estimates. A study conducted in the Navy, based on a database of avionics and sonar software data, attempted to investigate quantitative size estimating relationships in these systems. The result of this study [10] states that "no attribute factors have been found to be statistically significant for sizing adjustments, except language."

We conclude that much research needs to be done in order to have defined quantifiable system attributes upon which to base reliable size estimates. Despite the above criticism, the work of Abrecht, and Itakura and Takayanagi, represents a move from plain subjective expert estimation to more objective ways of quantifying software size early in the software development process. However, it seems clear that the goal they looked for needs to be pursued further yet.

Implementation Level Objective Models: Implementation level models attempt to express size as function of some characteristics which are more closely related to the program itself, whose values can be determined at detailed design, before actual coding begins. These quantities are, mainly, the number of operators and the number of unique variables and constants in the program. These models do not address our basic concern of early cost estimation. However they are included here for the sake of completeness of our discussions. A number of implementation level objective models for software size estimation have been proposed in the literature, such as the work of Fitsos [9] with the Software Science length equation, the work of Wang [37], and the work of Britcher and Gaffney [5]. Details of these models are provided by Levitin [16]–[18].

TABLE V
SOFTWARE SIZING FACTORS IN ITAKURA AND TAKAYANAGI'S MODEL

X1	number of input files
X2	number of input items
X3	number of output files
X4	number of output items
X5	number of items of transactions type records
X6	number of vertical items in two-dimensional table type reports
X7	number of horizontal items in two-dimensional table type reports
X8	number of calculating processes
X9	existence of sorting
X10	sum of output items in reports (X5 + X6 + X7)
X11	sum of output items in both files and reports (X4 + X5 + X6 + X7)

It is interesting to observe that, although more accurate estimates could be expected from using these models, due to more knowledge of the project which is available after the detailed design phase (Fig. 1), the results obtained by Levitin do not support this expectation. Her conclusion in [18] says that "program size cannot be estimated from the number of program variables with required accuracy. The problem is not our lack of knowledge of a right formula but rather the simple fact that programs with the same number of variables may differ considerably in their size." This conclusion is quite similar to ours in our previous considerations on specification level objective sizing models.

III. The Nature of Software Size Estimation

From the discussion carried out in the previous section we may say that the models and techniques available today for software sizing are not yet reliable enough to be consistently used with existing cost estimation models. On the one hand, subjective estimation, as it has been used in the software industry, lacks a solid enough basis to be able to provide sufficiently accurate sizing estimates. On the other hand, the attempts which aim to relate the size of a system to the number of certain external attributes of this system seem equally unable to be successful.

The task of software size estimation has been compared by Boehm [3] to the estimation of the number of pages of a novel before the novel is actually written. Let us suppose that a certain author is planning to write a novel which has four central characters, who influence each others' lives profoundly, and twenty incidental characters. The story happens in three different locations, it has a two year time span, and includes five detailed flashbacks. Suppose now that for some reason the editor wants to know in advance how many pages this novel will have after it is written. How could one estimate this quantity based solely on the given information? It is clear that the provided information is not enough to make possible such an estimate. In order to do that one would need additional information such as the complete relation of the events to take place in the novel, the detailed interrelationships between characters, the aspects on which the author wants to focus, etc. In other words, specific knowledge about the product would be necessary. Although the software sizing problem might not have all of the aspects of the novel sizing problem, the second gives us a good appreciation of the first.

The considerations so far presented seem to point out that specific knowledge about the nature of a software system which size one wants to estimate is a must. This includes knowledge of the system functions in terms of scope, complexity and interaction. Boehm studied the uncertainty in software project cost estimates as a function of the life cycle phase of a product [3]. The graphic in Fig. 2 shows the result of this study, which was empirically validated [3, Section 21.1] for a human-machine interface component of a program. The exponential curve in the figure resembles a "learning curve" and shows clearly that the uncertainty in cost estimates is related to the amount of knowledge (or lack of it) about the system at each phase of the software life cycle. Although the referred study was done with respect to cost estimation aspects of a project, the same concept can be easily extended to size estimation aspects.

With this in mind, it is not difficult to understand that subjective techniques fail because of lack of the necessary level of system knowledge, and the major flaw with objective models is that the factors they intend to base their estimates upon are not representative of all aspects of a system which might influence its final size.

If we want to have accurate software estimates we need new software sizing models which might be able to overcome the weaknesses of the existing ones. Furthermore, new models need also to solve the following problems:

1) We need specific knowledge of a system in its early stages of development.

2) We need to be able to relate, as accurately as possible, this knowledge to the physical size of the program.

3) Given that the ultimate level of information about a system in the early stages of its development is clearly

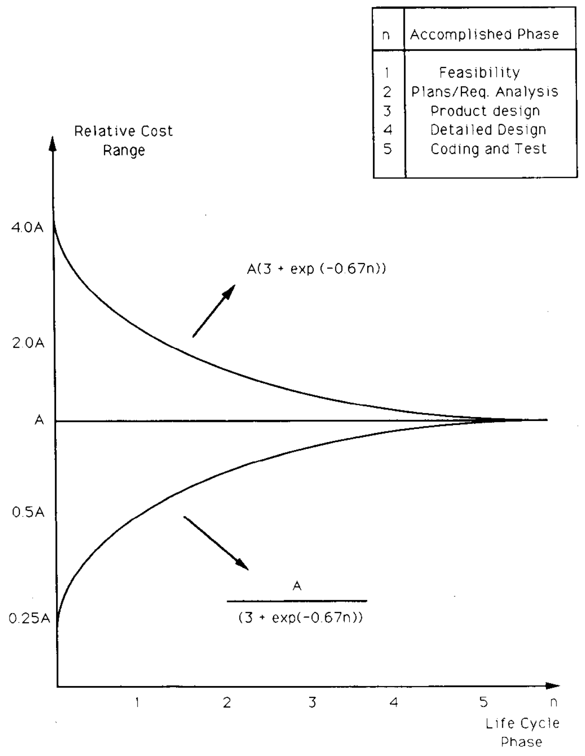

Fig. 2. Software project cost estimation accuracy as a function of the life cycle phase.

limited, we need to find a way to cope with this limitation and still have reliable sizing estimates, as well as be able to evaluate the degree of accuracy of the estimates.

The methodology we propose in this paper attempts to address these issues. In order to capture and represent knowledge about the system we use a functional requirements specification model, which enables decomposition of complexity and provides for understandability of its functionality. This model represents the system and associated environment as a number of semiautonomous objects acting asynchronously one upon another, just as the real world is. This representation points to the object-oriented approach for software development as a strong match to relate the knowledge embedded in the functional model to the final system size. This is so due to the fact that the representation of a real world problem with an object model will correspond to a great extent to the implementation of the corresponding system in an object-oriented programming environment. Finally, a statistical analysis provides for the necessary uncertainty evaluation. This analysis is based on the exponential curve of Fig. 2, adapted for the software sizing problem.

IV. The Object Model for Function Requirements and Software Size Estimation

Functional requirements are a must in any software system development in order to make it possible for developers and customers to understand the system they are both developing and proposing. The days of top-of-the-head functional specifications are already passed since the great majority of current large complex software systems demand a formal model to represent and document their functionality. Modeling the functionality of a software system is a task undertaken during the requirement analysis phase of the software life cycle.

The major role of a functional model is to provide for the understandability of the system it represents. The main problem here is how to decompose the system's complexity using a divide and conquer strategy. In [40], Yeh *et al.* point to three powerful and desirable properties of a functional model. They are as follows.

1) Abstraction: An abstraction represents several objects, suppressing details and concentrating on essential properties. It allows one to represent a system in several layers, called levels of abstraction [19], forming a natural hierarchy. Each level sees the level below as a virtual machine, with specified properties and functionality.

2) Partition: The partitioning of a system represents it as a sum of its parts, allowing one to concentrate on system components one at a time. High level components may also be partitioned, as well as its component parts. This leads to the idea of levels of partitioning. A component may also have an abstraction hierarchy. In this case a system would have both a horizontal and a vertical decomposition.

3) Projection: A projection of a system represents the entire system with respect to a set of its properties. This allows us to understand a system from different viewpoints, separating particular facets. An example of that could be the descriptive geometry description of a three-dimensional physical object which is composed of vertical and horizontal two-dimensional projections.

The object-oriented model for functional specification that we propose reflects these characteristics as we will discuss later. A number of researchers (Booch [4], Stark and Seidewitz [29]-[31]), have contributed to a general object-oriented methodology. Although they have concentrated somewhat more on design aspects of the object-oriented life cycle, the specification model we adopted here was largely influenced by their ideas.

In the proposed object model, each entity of the problem domain, or real world, is represented by an object. An object may be a person, a machine, a sensor, etc. Each object is composed by a state that characterizes it, and a set of functions, called methods, that manipulate the data corresponding to its state [32]. As a consequence, we may view an object as having two projections. One is the data structure that represents its state. The other is composed by the functions that determine its behavior. An object acts upon another by requesting the second object to perform a certain function on its data (state) and return some results to the requesting object. No object can act directly on another object's data. This concept is known as information hiding [22]. The fact that an object requests some action of another is represented in the model by an arrow directed from the object which requests the action to the object which performs the action. Therefore, an object is fully characterized by its state, its internal functions and its interactions with other objects (we will talk later about a fourth characteristic of an object—nonfunctional requirements it possibly needs to meet). We believe that a real world problem is easily represented by a set of objects that interact with one another.

At its top level a system may be represented by a single object that interacts with external objects. Beginning at this level each object of the system may be refined into component objects on a lower level of partitioning. This partitioning may continue until objects are completely decomposed into primitive objects. A primitive object would be one in which internal state (data) and corresponding methods (internal functions) are simple enough, in terms of complexity and size, to be considered an undecomposable entity of the system. The result of this process is a set of levels of partitioning that represents the system (Fig. 3). Furthermore, each partition may be viewed as a set of layers (or levels of abstraction). Each layer defines a virtual machine which provides service for senior layers. Any layer can request operations in junior layers but never in senior layers (Fig. 4). It should be noted that not all objects in a certain level of partitioning will be amenable for further partitions.

A specification effort should begin by identifying the entities in a problem domain and their interrelationships, and continue further by detailing the functions performed by and the internal state of each object. The next step would be to identify which objects could allow partitioning and the layers of abstraction in each partitioning level. A major advantage of such specification object model is that it makes possible a direct and natural correspondence with the real world, since problem domain entities are extracted directly into the model without any intermediate buffer such as traditional data flow diagrams [20]. This also makes the model quite understandable, which is an essential characteristic of a functional specification model.

The above considerations lead us to the fact that an object-oriented representation of a system is a more suitable model for accurate software size estimates than one achieved through a more traditional approach. The point here is that the implementation of the system, provided that it is also object-oriented, will match to a great extent its functional specification. This matching has been observed by Seidewitz in [31], where he reports that some projects designed and developed at Goddard Space Flight Center with an object-oriented approach, turned out to have a very smooth transition from specifications to code. We should notice that this matching does not necessarily occur with other specification methods such as common data flows, state transition diagrams, or data-oriented models. Although some of these models may also be used to specify systems in a hierarchical fashion, they lack the above mentioned correspondence between specification and implementation that favors the object-oriented model as a better choice for providing support for better software size estimates.

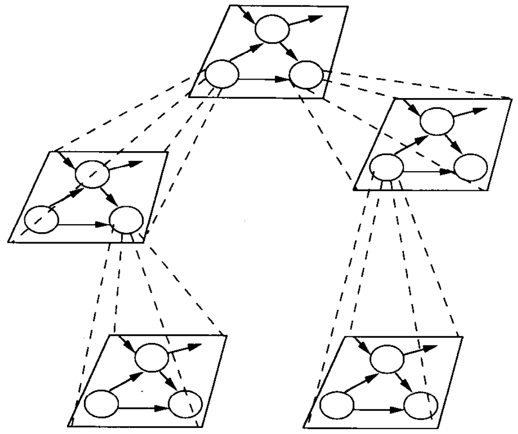
Fig. 3. Levels of decomposition (partitioning).

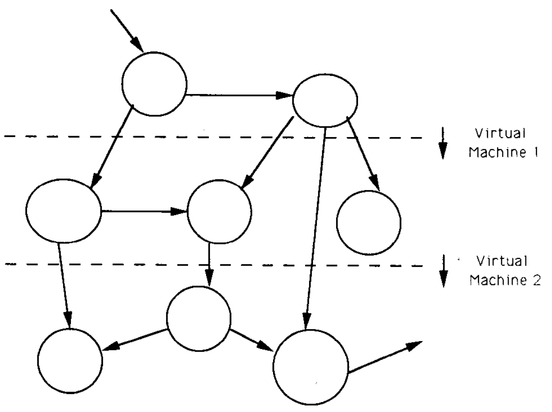

Fig. 4. Layers of abstraction.

We will not go into further detail about this object-oriented functional model, since our main goal here is to show its advantages for early software size estimation. In this scenario it is worth noting that we will be much more interested in the partitioning aspect of the objects of a system than in the aspect of layers of abstraction, although both aspects are needed when the specifications are derived. Whereas the second aspect is not needed for size estimation it will be very important concerning the development of the system.

The size estimation process of a system, whose functional model is characterized by a certain number of levels of decomposition (partitioning), may be summarized in the following steps:

1) Beginning with the lowest level of decomposition evaluate the size of each object. This evaluation should consider each function executed by the object, as well as the code corresponding to the data structures which will hold its internal state.

2) Continue to higher levels taking into account that higher level object sizes may receive contributions of component objects as well as of its own data and functions.

3) It may be necessary to include "utility objects" to account for housekeeping functions.

4) The estimated size of the whole system will be the sum of the size estimates of the objects in the top level of decomposition plus the size of possibly existing "utility objects."

It has been advocated in the literature [18] that one cannot expect to have reliable software size estimates based on specification models. Weinberg's experiment [38] is usually taken as the basis for this thesis. In that experiment, five programmer teams came up with considerably different program sizes for the same functional specifications. However, it should be said clearly that those teams were given quite different objectives during the development effort. One team was asked to minimize the amount of memory required for the program, another was to optimize program understandability, another was to minimize program length, another was to minimize the development effort, and the last one was to produce the clearest possible output. As it can be seen, although the general problem statement was the same for the five teams, they had quite different nonfunctional specifications. The conclusion we reach is that very restrictive nonfunctional specifications might influence software size. If some of them are present in a development, they should be considered in the size estimation process. This fact does not invalidate the sizing technique we present here, since our proposition is that estimates should be based on the available knowledge of the system. Whereas we consider project decomposition and functional specifications as major tools for detailing and representing system knowledge, other sources of knowledge such as nonfunctional requirements might also provide a contribution. A way to do that would be, for instance, to add to the characterization of each object a fourth element (other than data, methods or interactions) that would be called nonfunctional constraints. This would be a statement of restrictive nonfunctional requirements (if any) each object needs to fulfill. This information would also be used in the size estimation of that object.

Finally, we would like to point out that it is our belief that the most suited people for estimating software size are the future developers of the system. One reason for this is that each programmer has his own personal programming style, which often influences software size [17]. Another reason is that system developers, such as designers and programmers, are the people with the best conditions for accounting for the influence of nonfunctional specifications on system size.

V. Software Sizing: A Statistical Model

Fig. 2 shows a plot of the accuracy of software project estimates as a function of the software life cycle phase. This resulted from Boehm's study of projects in TRW. The meaning of this graph is that there is a very small probability that cost estimates will be out of the boundaries represented by the two converging exponential curves shown in the figure. If we put this in an analytic form we have

$$x(n) <= A * (3 + e^{-B*n}) \quad \text{(upper exponential)}$$

$$x(n) >= \frac{A}{3 + e^{-B*n}} \quad \text{(lower exponential).}$$

Here $x(n)$ is a cost estimate, A is the actual cost of the system, and n relates to the phase in the life cycle when the estimate was done. The value of B, in that experiment found to be 0.67, determines how fast estimates converge, that is, what is the improvement in the accuracy of estimates as we go from one phase to the next in the software life cycle.

Whereas Fig. 2 was primarily sketched with cost estimation in mind, we can extend the same concept to the problem of estimating software size. This extension is based on the fact that both cost and size estimates rely on the available knowledge of the system, as discussed before.

Fig. 5 shows the learning curve concept adapted to software size estimation. This figure differs from Fig. 2 in some aspects. Rather than relating size estimates throughout life cycle phases, it reflects the variation of size estimation accuracy, as one increasingly details the functional specification of the system by means of object decomposition, during the requirements analysis phase of the software development process. A feasibility phase is considered to be completed at this time. Therefore, the crossing points between the boundary exponentials and the vertical axis will be different from Fig. 2, where the feasibility phase is still considered. The value of n is now related to the level of decomposition of system objects in the specification based on which the estimate was done. In this scenario, $n = 0$ means that no functional specification was carried out yet, $n = 1$ means that a top level specification has been accomplished, $n = 2$ means that the objects in the top level were partitioned one level below, and so forth. The value of B accounts for how fast estimates converge to the actual size A, as we pursue in decomposing system objects down to lower and lower levels.

The hypothesis we base this upon is that the proposed object specification model will provide for a sufficiently disciplined methodology for capturing system knowledge as to cause estimates to converge smoothly to the actual size of system as further levels of object decomposition are reached. This means that an estimate made at a certain level of decomposition is not unrelated to previous estimates, obtained with less detailed levels of decomposition. This is easily seen if we consider that when going a level down in the decomposition process one just carries out the decomposition of objects in the current lower level. In this scenario, the amount of knowledge corresponding to undecomposed objects (not all objects will be decomposable) will remain the same, and the new knowledge achieved by the increased detail of decomposed objects will be a refinement of some less accurate knowledge that was already available before. The conclusion we reach is

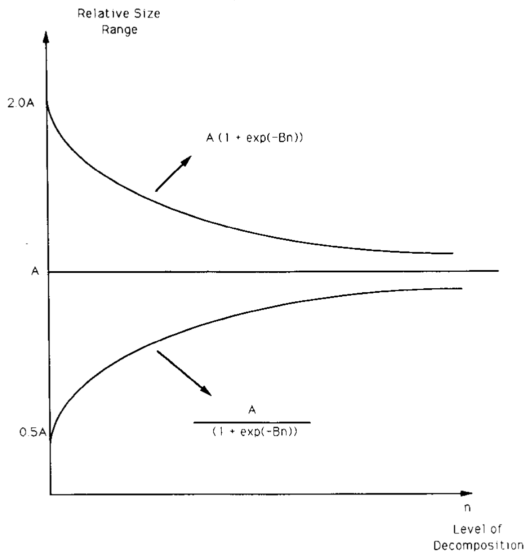

Fig. 5. Software size estimation accuracy as a function of object decomposition level in the functional model.

that, although extraneous estimates may show up here and there in the process, in general the pattern of estimates should converge monotonically to the actual size value. These considerations point to possible patterns of project size estimates as those shown in Fig. 6.

Additional important information, needed for the application of this technique, is how to obtain the value of B, which sets the rate for the exponential decay of upper and lower boundary curves of Fig. 5. After a lot of study and experimentation it is now well agreed upon among software engineering researchers and practitioners that any software metrics model needs to be tunned to the specific environment where it will be applied. In this work, the value of B is exactly the one which will incorporate with the estimating process the characteristics and previous experiences of the organization where a particular software project will take place. It should be calculated by analyzing a representative number of already accomplished projects with sizes estimated by using the proposed methodology. One would plot the values of project size estimates as a function of n, the level of decomposition of objects in the functional model, and look for the exponential boundary curves which are the "envelopes" for the entire range of estimates. This can be done by curve fitting statistical techniques.

Going further in the development of the model, we have that, for a particular estimate $x(n)$, evaluated considering n levels of objects decomposition, the following will hold

$$x(n) <= A * (1 + e^{-B*n}) \quad \text{(upper exponential)}$$

and

$$x(n) >= \frac{A}{1 + e^{-B*n}} \quad \text{(lower exponential).}$$

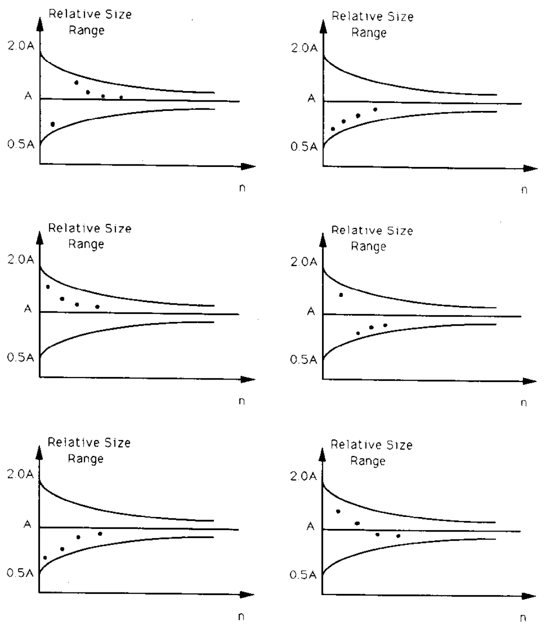

Fig. 6. Possible patterns for size estimates (assuming that subsequent estimates are not independent from one another).

These relations yield

$$A >= \frac{x(n)}{1 + e^{-B*n}} \quad \text{and} \quad A <= x(n) * (1 - e^{-B*n})$$

which can be rewritten as

$$A >= L1(n) \quad \text{and} \quad A <= L2(n)$$

where

$$L1(n) = \frac{x(n)}{1 + e^{-B*n}}$$

and

$$L2(n) = x(n) * (1 + e^{-B*n})$$

and finally,

$$L1(n) <= A <= L2(n).$$

Given B and n, we can also calculate a confidence interval for the estimates. This confidence interval turns out to be independent of A, the actual program size. The negative and positive deviations, $d1(n)$ and $d2(n)$, which characterize the confidence interval, can be calculated as

$$d1(n) = \frac{\dfrac{A}{1 + e^{-B*n}} - A}{A}$$

and

$$d2(n) = \frac{A * (1 + e^{-B*n}) - A}{A}$$

working with the above expressions we get

$$d1(n) = \frac{-e^{-B*n}}{1 + e^{-B*n}} \quad \text{and} \quad d2(n) = e^{-B*n}.$$

As $d1(n)$ and $d2(n)$ are independent of A, given a certain B we can plot their values as a function of n. Table VI shows the values of $d1(n)$ and $d2(n)$ for $B = 0.47$, with n varying from 1 to 6. We notice that the positive deviation is generally larger than the negative one, accounting for the biasing towards underestimation observed in reported software sizing experiences. The expected value for a size estimate given a certain $n = N$ will be

$$E(N) = \frac{L1_{\max}(n) + L2_{\min}(n)}{2}$$

where

$$L1_{\max}(N) = \max\,[L1(n)] \qquad \text{for } n <= N$$

and

$$L2_{\min}(N) = \min\,[L2(n)] \qquad \text{for } n <= N.$$

For $n = N$ there will be a very large probability that the actual program size A will be within the interval

$$[E_{\min}(N), E_{\max}(N)]$$

where

$$E_{\min}(N) = E(N) * (1 + d1(n))$$

and

$$E_{\max}(N) = E(N) * (1 + d2(n)).$$

It is interesting to note that, since the expected negative and positive deviations, $d1(n)$ and $d2(n)$, are independent of the actual size A, it is possible for one to know in advance the number of decomposition levels needed to achieve a desired accuracy. This fact provides the method with a criterion by which one can know whether it should or not be used in a particular project to be developed in a given organization. There will be cases when this technique will not be worth using. This will happen for small values of B, such that the convergence of the estimates with n will be so slow that, even if one gets to the lowest possible level of decomposition, the confidence interval will not satisfy the accuracy required for the particular project. The decision of using or not this technique will depend on the value of B, as well as on the required accuracy for a particular project cost estimation.

Detailing the method, we would have the following steps in the estimation process:

1) Given the value of B, plot in a table the values of $d1(n)$ and $d2(n)$ as a function of n.

2) Given the required accuracy for the estimation, check in that table what is the necessary level of object decomposition to go through in the functional specification model. Call it N.

3) Work the correspondent functional specifications

TABLE VI
PLOT OF NEGATIVE AND POSITIVE DEVIATIONS AS A FUNCTION OF n, FOR $B = 0.47$

n	d1(n)	d2(n)
1	-0.385	0.625
2	-0.281	0.391
3	-0.196	0.244
4	-0.133	0.153
5	-0.087	0.095
6	-0.056	0.060

one level of decomposition down. If this is the first step, model the system with top level objects.

4) Estimate system size as suggested in Section IV, yielding $x(n)$.

5) Calculate $L1(n)$, $L2(n)$, $L1_{\max}(n)$ and $L2_{\min}(n)$.

6) Calculate the expected system size $E(n)$, and the maximum and minimum expected sizes, $E_{\max}(n)$ and $E_{\min}(n)$, for this level of decomposition n.

7) If $n = N$ stop. Consider $E_{\min}(N)$, $E(N)$, and $E_{\max}(N)$ as the minimum, expected, and maximum values for system size. If $n < N$ and the objects in the functional model cannot be decomposed one more level, stop and disregard results so far achieved. The method will not provide enough accuracy in this case. If objects still allow decomposition, go one step further to item (4).

As the proposed method is very sensitive to the value of B, it is important to consider that this value will not be static for a particular environment. In other words, estimators may "learn" how to come up with better estimates by getting more used to the methodology and the environment characteristics, and by relying more and more on the experiences of past projects. With this in mind, it would be wise to recalculate B when comparison between estimates and actual size values shows that the method is yielding too broad confidence intervals in face of the increased accuracy of estimates.

We can illustrate the application of the proposed technique with two examples. Project I is a relatively small business application. The actual size of the system was found to be 10,000 lines of code, after the project was accomplished. The value of B for the corresponding organization has been evaluated as 0.47, and, as the entire cost of the project is relatively low, an overall accuracy of 25% was considered acceptable. Looking at Table VI we notice that we will need up to three levels of decomposition in the functional model. Table VII shows the values of the quantities calculated in the estimating process, and Fig. 7 presents a graph of the expected size $E(n)$ provided by the model. We can see that the first estimate shows a gross discrepancy with respect to the actual achieved size, but subsequent ones converge smoothly to the value of A. By the end of the process the expected size in lines of code is 10,590, with possible deviation values of -2076 and $+2584$.

TABLE VII
SIZE ESTIMATION CALCULATIONS FOR PROJECT I, WITH $B = 0.47$

n	x(n)	L1(n)	L2(n)	$L1_{\max}(n)$	$L2_{\min}(n)$	E(n)	$E_{\min}(n)$	$E_{\max}(n)$
1	7,000	4,308	11,375	4,308	11,375	7,842	-3,019	4,091
2	13,000	9,348	18,078	9,348	11,375	10,362	-2,912	4,052
3	12,200	9,806	15,174	9,806	11,375	10,590	-2,076	2,584

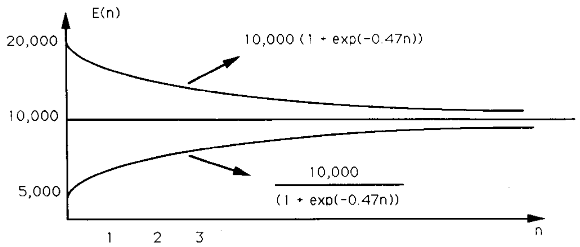

Fig. 7. Learning curve for Project I size estimates $E(n)$.

Project II, carried out in the same company, corresponds to the development of an operating system. This project required an accuracy of 10% due to organization budget constraints at the time. Since B is the same as in Project I, Table VI shows that functional specifications should go through 5 levels of object decomposition. Table VIII presents the values calculated in the estimating process. The expected size value was 37,505 lines of code, and deviation values -3263 and $+3563$. As actual program size was found to be 40,000 lines of code, we can see in Fig. 8 that model estimates $E(n)$ converge smoothly to A as n increases.

VI. RELATING SIZE TO COST

As seen in the previous section, the size estimation method proposed in this paper allows one to estimate system size with a specified confidence interval. This confidence interval depends on the value of B, the exponential decay of the size estimates exponential "envelope" curve, and on the value of n, the level of object decomposition in the functional specifications model. Since our ultimate goal in estimating program size is to be able to predict program cost as accurately as possible, it would be nice, now, to relate our size estimation results to cost. In other words, we would like to know how much to budget for a certain software project and what level of variance should be expected from this estimation. In order to do that we can use, for instance, the COCOMO model [3] to calculate the estimated development cost of Project I and Project II, the examples of the previous section.

COCOMO predicts project cost, in man-months, primarily based on the estimated number of thousands of lines of code for the system. This nominal estimation is then adjusted by a number of effort multipliers, whose overall product is called Effort Adjustment Factor (EAF). COCOMO effort multipliers are summarized in Table IX. Observation of the nature of effort multipliers shows that the necessary system knowledge for their evaluation is

TABLE VIII
SIZE ESTIMATION CALCULATIONS FOR PROJECT II, WITH $B = 0.47$

n	x(n)	L1(n)	L2(n)	L1 (n) max	L2 (n) min	E(n)	E (n) min	E (n) max
1	28,000	17,231	45,500	17,231	45,500	31,366	-12,076	19,604
2	32,500	23,011	45,901	23,011	45,901	34,456	-9,682	13,472
3	34,700	27,891	43,712	27,891	43,172	35,532	-6,964	8,670
4	35,200	30,540	40,571	30,540	40,571	37,885	-5,039	5,796
5	37,350	34,098	40,912	34,098	40,912	37,505	-3,263	3,563

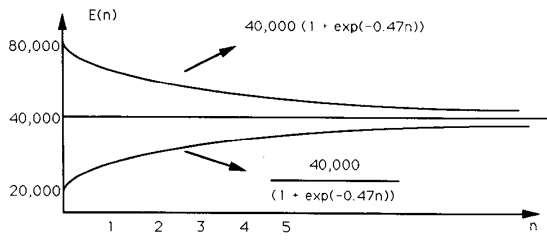

Fig. 8. Learning curve for Project II size estimates $E(n)$.

certainly available by the requirements/specifications phase of the software life cycle, i.e., at $n = 0$.

For these examples, as we use COCOMO, our assumptions will be rather simple, since our purpose here is not to detail COCOMO itself, but to relate the presented size prediction method to cost estimation. In this scenario, we can consider the top level objects in the functional model as the system components referred to in Intermediate COCOMO. For the sake of simplicity we assume that effort multipliers are the same for all system components (top level objects) in Project I and Project II. This assumption allows us to view each of these objects, for cost estimation purposes, as having just one component. We also assume that both systems are developed from scratch. Another point is that the development team has extensive experience in working with related projects, allowing us to classify these projects as organic, in COCOMO terminology [3]. Intermediate COCOMO formulas for effort estimation of organic projects are

$$(\mathrm{MM})_{\mathrm{nom}} = 3.2 * (\mathrm{KEDSI})^{1.05}$$

$$(\mathrm{MM})_{\mathrm{adj}} = (\mathrm{MM})_{\mathrm{nom}} * \mathrm{EAF}$$

$$\mathrm{EAF} = \Pi \ \mathrm{EM}_i$$

where KEDSI (thousands of expected delivered source instructions) is the estimated program size in thousands of lines of code, and the EM_i's are the effort multipliers.

Table X and Table XI show the calculated effort adjustment factor (EAF) for Project I and Project II, respectively. Table XII and Table XIII summarize the cost estimation process for Project I and Project II, respectively. The KEDSI values in these tables estimated using the proposed sizing method. In order to express project cost in dollars, the average equivalence in dollars per man-month was also considered for each project.

TABLE IX
COCOMO EFFORT MULTIPLIERS

EM	Significance	EM	Significance
RELY	Required Soft. Reliability	AEXP	Application Experience
DATA	Data Base Size	PCAP	Programmers Capability
CPLX	Product Complexity	VEXP	Virtual Machine Experience
TIME	Execution Time Constraint	LEXP	Prog. Language Experience
STOR	Main Storage Constraint	MODP	Use of Modern Prog. Practice
VIRT	Virtual Machine Volatility	TOOL	Use of Software Tools
TURN	Computer Turnaround Time	SCED	Required Development Schedule
ACAP	Analysts Capability		

TABLE X
EFFORT ADJUSTMENT FACTOR CALCULATION FOR PROJECT I

EM	Rating	Value
RELY	Nominal	1.0
DATA	Low	0.94
CPLX	Nominal	1.0
TIME	Nominal	1.0
STOR	Nominal	1.0
VIRT	Nominal	1.0
TURN	Low	0.87
ACAP	High	0.86
AEXP	Nominal	1.0
PCAP	High	0.86
VEXP	Nominal	1.0
LEXP	Nominal	1.0
MODP	Low	1.10
TOOL	Nominal	1.0
SCED	Nominal	1.0
EAF		0.67

TABLE XI
EFFORT ADJUSTMENT FACTOR CALCULATION FOR PROJECT II

EM	Rating	Value
RELY	Very High	1.40
DATA	Low	0.94
CPLX	Nominal	1.0
TIME	High	1.11
STOR	Nominal	1.0
VIRT	Nominal	1.0
TURN	Nominal	1.0
ACAP	High	0.86
AEXP	High	0.91
PCAP	Nominal	1.0
VEXP	Nominal	1.0
LEXP	Nominal	1.0
MODP	Nominal	1.0
TOOL	Nominal	1.0
SCED	Nominal	1.0
EAF		1.14

TABLE XII
COST ESTIMATION FOR PROJECT I

Component	KEDSI	EAF	MM nom	MM adj	$K/MM	Cost ($K)
Text Processing System	Minimum 8.51	0.67	30	20	5.0	Minimum 101.54
	Expected 10.59	0.67	38	26	5.0	Expected 127.74
	Maximum 13.71	0.67	48	32	5.0	Maximum 160.61

The size estimation technique proposed in this paper enables us to say that the ultimate size of Project I will be 10.59 KEDSI, with possible deviation interval of -2.08 KEDSI, $+2.58$ KEDSI. For Project II the expected final size is 37.51 KEDSI, with possible deviation interval of -3.26 KEDSI, $+3.56$ KEDSI.

As a consequence, we may also say that the cost of Project I will be \$127,740 (dollars), with possible devia-

TABLE XIII
COST ESTIMATION FOR PROJECT II

Component	KEDSI	EAF	MM nom	MM adj	$K/MM	Cost ($K)
Operating System	Minimum 34.24	1.14	131	149	5.5	Minimum 819.74
	Expected 37.51	1.14	144	164	5.5	Expected 902.14
	Maximum 41.07	1.14	158	180	5.5	Maximum 992.24

tion interval of −$26,200 (dollars), +$32,870 (dollars), and the cost of Project II will be $902,140 (dollars), with possible deviation interval of −$82,400 (dollars), +$90,100 (dollars). This type of follow-through of dollarizing fulfills the manager's needs of cost and risk assessment.

VII. Concluding Remarks

A technique for software size estimation has been proposed. The basis for estimates, when using this method, is the available knowledge of the considered system. In order to capture and represent this knowledge, an object-oriented functional model has been adopted. This functional model provides for a disciplined methodology for decomposing system complexity. This methodology is the key in the process of detailing the functionality of the system in order to enable estimators to achieve more reliable estimates. The object-oriented paradigm plays an important role in this process since it embeds a strong correspondence between specifications and implementation. This characteristic makes it easier to relate an object functions, sometimes called methods, and data to the amount of code necessary to implement it.

In order to relate previous experience with size estimation in a certain organization, the proposed sizing technique incorporates a statistical approach. This approach also enables one to have an objectively derived confidence interval for the estimates, what has been a desire among software metrics researchers.

The presented methodology is still subjective, since it ultimately depends on expert estimations. Nevertheless, this subjectiveness is controlled by disciplined capturing of system knowledge and statistical correlation with past experience. The methodology also provides a criterion that enables one to know when the amount of subjectiveness related to the estimates prevents its use. Also, certain issues like nonfunctional requirements influence and low biasing in software estimations have been considered in this sizing method.

Finally, we saw that the utilization of this sizing technique with cost estimation models enables these models to predict system cost with known accuracy, what provides for better controlled and managed software projects.

Acknowledgment

We are thankful to Dr. G. Cobb (with Lockheed, Austin), for various insights concerning the ideas upon which this paper is built, and to the reviewers, for their helpful suggestions.

References

[1] A. J. Albrecht and J. E. Gaffney, "Software function, source lines of code, and development effort prediction: A software science validation," *IEEE Trans. Software Eng.*, vol. SE-9, no. 6, pp. 639-647, Nov. 1983.
[2] J. W. Bailey and V. R. Basili, "A meta-model for source development resource expenditures," in *Proc. Fifth Int. Conf. Software Engineering*, 1981, pp. 107-116.
[3] B. Boehm, *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.
[4] G. Booch, "Object-oriented development," *IEEE Trans. Software Eng.*, vol. SE-12, no. 2, pp. 211-221, Feb. 1986.
[5] R. N. Britcher and F. E. Gaffney, "Reliable size estimation for software decomposed as state machines," in *Proc. COMPSAC 1985*, pp. 104-106.
[6] A. Bryant and J. A. Kirkham, "B. A. Boehm software engineering economics: A review essay," *ACM SIGSOFT Software Eng. Notes*, vol. 8, no. 3, pp. 44-60, July 1983.
[7] S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models.* New York: Benjamin/Cummings, 1986.
[8] F. R. Freiman and R. E. Park, "Price software model version 3: An overview," in *Proc. IEEE-PINY Workshop Quantitative Software Models*, Oct. 1979, pp. 32-41.
[9] G. P. Fitsos, "Vocabulary effects in software science," in *Proc. COMPSAC 1980*, pp. 751-756.
[10] S. Gross, K. B. Tom, and E. E. Ayers, "Software sizing and cost estimation study," in *Proc. 19th Annu. Dep. Defense Cost Analysis Symp.*, Xerox Training Center, Leesburg, VA, Sept. 17-20, 1985.
[11] J. R. Herd, J. N. Postak, W. E. Russel, and K. R. Stewart, "Software cost estimation study—Study results," Doty Associates, Inc., Rockville, MD, Final Tech. Rep. RADC-TR-77-220, June 1977.
[12] M. Itakura and A. Takayanagi, "A model for estimating program size and its evaluation," in *Proc. Sixth Int. Conf. Software Engineering*, Sept. 1982, pp. 104-109.
[13] K. A. Jamsa, "Object-oriented design vs structured design—A student's perspective," *ACM SIGSOFT Software Eng. Notes*, vol. 9, no. 1, pp. 43-49, Jan. 1984.
[14] R. W. Jensen, "A comparison of the Jensen and COCOMO schedule and cost estimation models," in *Proc. Int. Soc. Parametric Analysis*, 1984, pp. 96-106.
[15] L. Ledbetter and B. Cox, "Software ICs," *Byte*, vol. 10, no. 6, pp. 307-316, June 1985.
[16] A. V. Levitin, "On predicting program size by program vocabulary," in *Proc. IEEE COMPSAC 1985*, pp. 98-103.
[17] —, "How to measure program size and how not to," in *Proc. IEEE COMPSAC 1986*, pp. 314-318.
[18] —, "Investigating predictability of program size," in *Proc. IEEE COMPSAC 1987*, pp. 231-235.
[19] B. Liskov and J. Guttag, *Abstraction and Specification in Program Development.* Cambridge, MA: MIT Press, 1986.
[20] T. de Marco, *Structured Analysis and System Specification.* Englewood Cliffs, NJ: Prentice-Hall, 1979.
[21] D. L. Parnas, "A technique for the specification of software modules with examples," *Commun. ACM*, vol. 15, pp. 330-336, May 1972.
[22] —, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, pp. 1053-1058, Dec. 1972.
[23] G. A. Pascoe, "Elements of object-oriented programming," *Byte*, vol. 11, no. 8, pp. 139-144, Aug. 1986.
[24] R. Pressman, *Software Engineering: A Practitioner's Approach.* New York: McGraw-Hill, 1982.
[25] L. H. Putman, "A general empirical solution to the macro software sizing and estimation problem," *IEEE Trans. Software Eng.*, vol. SE-4, no. 4, pp. 345-361, July 1978.
[26] L. H. Putnam and A. Fitzsimmons, "Estimating software costs," *Datamation*, pp. 189-198, Sept. 1979; continued in *Datamation*, pp. 171-178, Oct. 1979, pp. 137-140, Nov. 1979.
[27] C. V. Ramamoorthy, A. Prakash, W. Tsai, and Y. Usuda, "Software engineering: Problems and perspectives," *Computer*, pp. 191-209, Oct. 1984.
[28] D. T. Ross and K. E. Scoman, "Structured analysis for requirements definition," *IEEE Trans. Software Eng.*, vol. SE-3, no. 1, Jan. 1977.
[29] M. Stark and E. Seidewitz, "Towards a general object-oriented de-

velopment methodology," Goddard Space Flight Center, Greenbelt, MD, Internal Rep., 1986.
[30] ——, "Towards a general object-oriented ADA life-cycle," in *Proc. Joint Ada Conf.*, Mar. 1987, pp. 213-222.
[31] E. Seidewitz, "General object-oriented software development: Background and experience," in *Proc. 21st Hawaii Int. Conf. System Science*, Jan. 1988.
[32] M. Stefik, "Object-oriented programming: Themes and variations," *AI Mag.*, pp. 40-62, Jan. 1986.
[33] C. R. Symons, "Function point analysis: Difficulties and improvements," *IEEE Trans. Software Eng.*, vol. 14, no. 1, pp. 2-11, Jan. 1988.
[34] R. C. Tausworthe, "Deep space network software cost estimation model," Jet Propulsion Lab., Pasadena, CA, Publ. 81-7, 1981.
[35] L. Tesler, "Programming experiences (with object-oriented languages)," *Byte*, vol. 11, no. 8, pp. 195-206, Aug. 1986.
[36] C. E. Walston and C. P. Felix, "A method of programming measurement and estimation," *IBM Syst. J.*, vol. 16, no. 1, pp. 54-73, 1977.
[37] A. S. Wang, "The estimation of software size and effort: An approach based on the evolution of software metrics," Ph.D. dissertation, Dep. Comput. Sci., Purdue Univ., Aug. 1984.
[38] G. W. Weinberg and E. L. Schulman, "Goals and performance in computer programming," *Human Factors*, vol. 16, no. 1, pp. 70-77, 1974.
[39] R. W. Wolverton, "Software costing," in *Software Engineering*, C. R. Vick and C. V. Ramamoorthy, Eds. New York: Van Nostrand, 1981.
[40] R. T. Yeh, P. Zave, A. P. Conn, and G. E. Cole, Jr., "Software requirements: New directions and perspectives," in *Software Engineering*, C. R. Vick and C. V. Ramamoorthy, Eds. New York: Van Nostrand, 1981.

Luiz A. Laranjeira (S'89) was born in Belo Horizonte, Brazil, in 1958. He received the B.S. degree from the University of Brasilia, Brasilia, Brazil, in 1979, and the M.Sc. degree from the Federal University of Rio de Janeiro, Rio de Janeiro, Brazil, in 1983, both in electrical engineering.

From 1984 until 1987 he was with the Research and Development Center for Communications, Brasilia, as Software Manager of the Division of Digital Projects. Since 1987 he has been a graduate student with the Department of Electrical and Computer Engineering of the University of Texas at Austin. His research interests are software engineering, parallel algorithms, and fault tolerance.

Mr. Laranjeira is a member of Tau Beta Pi.