An Empirical Test of Object-based Output Measurement Metrics in a Computer Aided Software Engineering (CASE) Environment

RAJIV D. BANKER, ROBERT J. KAUFFMAN,
AND RACHNA KUMAR

RAJIV D. BANKER holds the Arthur Andersen Chair in Accounting and Information Systems at the Carlson School of Management, University of Minnesota. He received a doctorate from Harvard Business School, specializing in planning and control systems. He currently serves on the editorial boards of six journals and as coeditor of the *Journal of Productivity Analysis*. He has published over forty refereed articles. His research interests include strategic cost management, measuring the business value of information technology, assessing software development and maintenance productivity, and the economics of information.

ROBERT J. KAUFFMAN is an Assistant Professor at the Stern School of Business at New York University, where he has taught since 1988. He completed his masters degree in international affairs at Cornell University, and was later employed as an international lending and strategic planning officer at a large money center bank in New York City. He received a doctorate in information systems from the Graduate School of Industrial Administration at Carnegie Mellon University in 1988. His current program of research involves developing new methodologies for measuring the business value of a broad spectrum of information technologies, using techniques from management science and economics. He has published refereed articles in *Journal of Management Information Systems, Information and Software Technologies, MIS Quarterly,* and elsewhere.

RACHNA KUMAR is currently in the doctoral program in information systems at the Stern School of Business, New York University. She received an M.S. in physics and

an MBA from the Indian Institute of Management at Ahmedabad, India, in 1983. Her current research interests focus on productivity measurement and cost estimation for computer-aided software engineering (CASE) environments. Her dissertation work involves a field study of the performance of object-based productivity metrics in the various CASE life-cycle phases.

ABSTRACT: Existing output measurement metrics for cost estimation and development productivity need to be reexamined to determine their performance in computer-aided software engineering (CASE) development environments. This paper critiques and empirically evaluates four approaches to the measurement of outputs. Two of the metrics, *raw function counts* and *function points*, are based on the function point analysis methodology pioneered by Albrecht and Gaffney at IBM. The second two, *object counts* and *object points*, are based on a new approach—object points analysis—that is introduced here for the first time. The latter metrics are specialized for output measurement in object-based CASE environments that include a centralized object repository. Estimation results for nineteen large-scale CASE projects show that the new metrics have the potential to yield equally accurate, yet easier to obtain estimates than function points-based measures.

1. Introduction

THE PRODUCTIVITY IMPACTS AND BUSINESS VALUE implications of computer-aided software engineering (CASE) tools are of increasing concern to information systems researchers and practitioners in the software development community. However, convincing results in this area have been difficult to obtain [4, 21]. The lack of results can be attributed to a number of difficulties ranging from poor data availability to limitations of current evaluation approaches [15, 17]. Thus, there is substantial motivation to conduct research on measurement approaches that are conducive to building a cumulative base of valid and reliable estimates for the outputs and process of CASE development.

1.1. The Research Problem: Estimation and Productivity Assessment in CASE Environments

A survey recently conducted by *Software Magazine* reported that only 13 percent of the firms in a sample of 196 CASE-using firms surveyed had a productivity measurement program of any kind in place [6]. Surveys such as this one indicate the need for measurement approaches that identify and substantiate CASE-related productivity improvements. Appropriate measurement approaches will not only allow comparisons across different CASE development environments, they will also increase the effectiveness of management control systems that aim to improve strategic cost management by more carefully tracking software development productivity [3].

However, before measurement can proceed, robust metrics must be established as measurement units. Existing measurement approaches were developed and validated for third-generation language (3GL) software development environments. The CASE development environment, however, differs in two ways [25]:

* *Structurally*, since systems can reuse the designs and functionality of existing systems through reusable software modules and routines; and,
* *Functionally*, since the tools that support CASE software development are quite different from those used in traditional development, and actually change the process itself.

Thus, although well-established methods should be used to improve our understanding of CASE productivity, they also must be scrutinized and, if necessary, recalibrated to ensure they remain valid under a new set of development conditions.

This paper examines the issue of output measurement for object-based software in a computer-aided software engineering environment. It critiques and empirically evaluates four approaches to the measurement of outputs. Two of the metrics, *raw function counts* and *function points*, are based on the *function point analysis* methodology developed by Albrecht and Gaffney at IBM. Function points measure the intrinsic size of the outputs of software development [1]. The second two, *object counts* and *object points*, are based on a new approach—*object points analysis*—that is introduced here for the first time. This approach involves counting software objects that have been developed. The metrics are specialized for output measurement in object-based CASE environments that include a centralized object repository. The central premises of these four metrics are reviewed in Table 1.

We present estimation performance results of the four alternative metrics in terms of their ability to predict software development effort. *Estimation performance* in this research refers to the ability of a software output measurement metric to accurately predict the amount of software development labor consumed in a project. This will enable us to assess the extent to which each of the metrics actually measures the size of the software. Our results show that the new metrics have the potential to yield equally accurate, yet easier to obtain estimates than function points-based measures.

1.2. Function Points as an Output Metric

Function points is a metric for the size of the output delivered by the software development process. A function point is defined as one end-user business function [1]. This metric was originally employed as a means to track productivity, which is usually measured in terms of function points delivered per person-month of development effort. Subsequent research has investigated the ability of *a priori* estimates of function points to *predict* the effort required for developing software, and function points repeatedly has been shown to be a good estimator [14, 18, 24].

The function point analysis procedure requires the analyst to identify the occurrences of each of five unique function types. These include *External Inputs, External Outputs, Logical Internal Files, External Interfaces*, and *Queries* delivered by the software. For the purposes of this research, we call the sum of all function type

Table 1   Four Output Metrics for CASE Cost Estimation and Development
Productivity Measurement

METRIC		DESCRIPTION OF CENTRAL PREMISES OF PROPOSED METRIC
RAW-FUNCTION-COUNTS	*	Represents a simple count of the five function types from function point analysis: Inputs, Outputs, Queries, Interfaces and Files.
	*	Function type weighting and environmental complexity measures associated with FUNCTION-POINTS may lead to over-estimates of labor required.
	*	CASE development tends to make most tasks less labor-intensive.
	*	This metric differs from FUNCTION-COUNTS in function point analysis; FUNCTION-COUNTS incorporates multiple levels of complexity, according to which each function type are weighted.
FUNCTION-POINTS	*	Recognizes that functionality, as opposed to source-lines-of-code, may provide best estimate of development effort consumed.
	*	Five primitive function types common to all software were proposed by Albrecht
	*	Function types are weighted according to relative complexity in a given application. Weighted scores are summed, and adjusted by an environmental complexity score, resulting in FUNCTION-POINTS.
	*	FUNCTION-POINTS is the base case for this analysis.
OBJECT-COUNTS	*	Analogous to RAW-FUNCTION-COUNTS in function point analysis.
	*	Represents a simple count of all objects in application's object hierarchy stored in repository.
	*	Objects in object-based CASE development environment offer a conceptually simple measure of functionality.
	*	Results of our field study suggest that CASE development methods tend to reduce relative complexity of creating software functionality.
	*	Objects only counted, not weighted to distinguish different levels of functionality that would require different levels of labor.
OBJECT-POINTS	*	Analogous to FUNCTION-POINTS, however, utilizes weighted OBJECT-COUNTS instead of FUNCTION-COUNTS.
	*	Weights applied to OBJECT-COUNTS were determined based on extensive project manager interviews and group estimation sessions.
	*	Managers interviewed rejected premise that weighted object estimates required further adjustment to represent environmental complexity of CASE development.

occurrences the *RAW-FUNCTION-COUNTS* (RFC). In a standard function point analysis, however, this number is not used. Instead, instances of each function type are identified and then weighted with numbers that reflect the level of development complexity of a given function type. These weighted values are then summed to arrive at *FUNCTION-COUNTS* (FC). FUNCTION-COUNTS is then adjusted using ratings on fourteen complexity factors that reflect the complexity of the system requirements and the overall software development environment. The adjustment factor is called the *TECHNICAL-COMPLEXITY-FACTOR* (TCF). Finally, *FUNCTION-POINTS* (FP) is calculated as FUNCTION-COUNTS * TECHNICAL-COMPLEXITY-FAC-TOR. (Appendix 1 gives further details about the content of the complexity levels that lead to FUNCTION-COUNTS and the characteristics that describe the environmental complexity captured by the TECHNICAL-COMPLEXITY-FACTOR.)

A number of reasons support the choice of function points as the primary measurement approach to be evaluated. First, the function points metric is widely accepted as a *de facto* industry standard [1, 12, 18, 26]. Although there are a variety of approaches to counting function points, including the ESTIMACS [23], SPQR [12], MARK II [26], International Function Point Users Group (IFPUG) [11], and IBM [10] standards, generally the rules for counting function points have been rigorously defined and agreed upon by their more enthusiastic users [7, 11]. This is especially true for the IFPUG standard. For example, *Software Magazine* recently reported on the sharp increase in IFPUG's membership, and the contribution that the organization has made toward the increasing standardization of the measurement of function points [17].

Second, function points also has advantages over source-lines-of-code methods of software output size estimation. Function points can be estimated earlier in the development cycle, and are independent of the language and technology used [1, 13, 18]. Kemerer [14] reports that function points led to a smaller average error rate in estimating software applications when compared to alternate output measurement methods, including the popular source-lines-of-code-based models, COCOMO and SLIM, and ESTIMACS.

Third, a major concern is that measures for software development outputs be robust across different people who make the estimates and across different estimation methods. More recent research by Kemerer [16] suggests that function points meets both of these requirements. He showed that function points are reliable within plus or minus 10 percent under both circumstances. Apparently the market has already recognized this, since most CASE customers with measurement plans are basing their productivity metrics on function points [6].

1.3. Data and the CASE Environment Examined in the Study

We obtained data on nineteen projects from a large investment bank in New York City. The projects were developed and implemented with CASE over a two-year period. Table 2 presents an overview of some representative projects from this sample.

The CASE tool that was used to develop these applications evolved as a multimillion dollar, internally developed software project. Its objective was to increase the responsiveness of the firm's software development operations, and to reduce the risk that whatever software was built would become rapidly obsolete. The cornerstone of the firm's software development strategy was to promote software reusability. (For additional details of the firm's strategy, see [4].)

The firm's CASE tool set exhibits many of the features of an Integrated CASE Environment (ICE) [2]. In this research, we will use the term ICE to refer to application development using CASE tools that automate a set of activities that span the entire life cycle of software development. Such automation begins with tools to support the earlier stages of analysis and design, and continues into the later stages of code construction and testing. As a result, ICE allows for the reuse of designs and code in primary development, as well as in maintenance.

The type of CASE environment present when applications are developed dictates

Table 2   An Overview of Software Projects Developed Using ICE*

APPLICATION	DESCRIPTION
Dealer's Clearance	Designed to improve operational and treasury management productivity by automating settlement, providing on-line, real-time display of clearances, and projected end-of-day securities and cash balance positions.
General Ledger Interface	A table-driven, self-balancing system that automatically posts entries from every transaction processing system included in NTPA. As a result, manual reconciliations are never required.
Firm Inventory/ Foreign Securities & Currencies	Maintains information for firm-wide management of foreign securities and currencies. Tracks individual trade lots and can determine profit and loss using various trading accounting bases.
Floor Broker	Manages fee and discount information for all brokers used by the firm. The system maintains payment histories linked to exchange, broker and trading volume.
Product Master	This system supports identification of financial products across business areas. It enables each business group to classify and process securities according to its own business requirements, and it allows trading areas to establish new product types in the process of conducting business.
Real-time Firm Inventory	Trading management uses this system to monitor trading positions, exposures, and intraday profit and loss by product, account, desk, department, or firm-wide. This system also enables traders to set up and monitor a strategy by linking several positions.

* Adapted from [4].

the variety and range of automated software engineering facilities available for programming. ICE provides powerful development-support utilities, including entity-relationship modeling, screen and report painters, and 3GL module-integration tools. Its unique features include:

• An *object-based* approach to applications development. Application programmers use structured, standardized, and rigorously defined objects and modules as building blocks to encode the functionality required for applications;
• A *centralized repository* that stores all modules and objects developed for applications;
• Storage of the application's structure as an abstract *object hierarchy* in the repository. This high-level structural representation of the application defines the relationships among the objects that deliver the functionality of each ICE application.

The remainder of this paper is organized as follows. Section 2 critiques function point analysis from the perspective of development in CASE environments. It also

discusses our rationale for testing the RAW-FUNCTION-COUNT metric for CASE-developed systems, as a short-form variation of FUNCTION-POINTS. Section 3 presents a new approach to gauging the outputs of software development for object-based CASE environments: object points analysis. The OBJECT-COUNTS and OBJECT-POINTS metrics are discussed in detail. Then, in section 4, the results of our empirical evaluation of the four output metrics are presented. Section 5 concludes with a consideration of the requirements that must be met for software output size metrics to better support the measurement and estimation of productivity in systems developed using CASE.

2. Function Points from a CASE Perspective: A Critique

How do FUNCTION POINTS STAND UP to the challenge of measuring software output size in an object-based CASE environment? What portions of the function point analysis procedure present problems that can be overcome using revised metrics? In this section, we will argue that each step in the calculation of function points (as presented in appendix 1) needs to be reassessed in light of relevant CASE characteristics.

2.1. Step 1—Identification of Function Types

First, the classification scheme used in the identification of the five function types is not intuitive for CASE-developed software. The components of the function points procedure (external inputs, external outputs, external interfaces, queries, and logical internal files) do not follow naturally from the building blocks of an object-based CASE environment such as ICE. In this development environment the objects themselves define the functionality of the application. This is shown in the high-level structure chart of a typical ICE application presented in figure 1.

The CASE methodology used in object-based ICE development also enforces modularization of application code. When modules and objects are the building blocks of CASE applications, identification of the five function types will force the analyst to expend significant effort to examine the code associated with a module or an object. Moreover, a sizeable portion of the code may have originated from a powerful feature of CASE: the ability to generate code. A programmer or analyst who has not written the actual code and done only the logical design would be forced to deal with the automatically generated code. Such code may not closely match what the person would have written. Thus, analyzing CASE-generated code would be an onerous, and, most likely, an inefficient task. This process would likely result in subjectivity and inconsistency in the classification of the function types, as well as require a large amount of time and effort on the part of the analyst.

Second, a straightforward gauge of function types will be prone to double-counting the labor consumed in developing systems with CASE. The central repository in ICE offers the developer significant opportunities to reuse code [22]. Reused code adds to the functionality a system delivers without requiring much additional effort. So, when
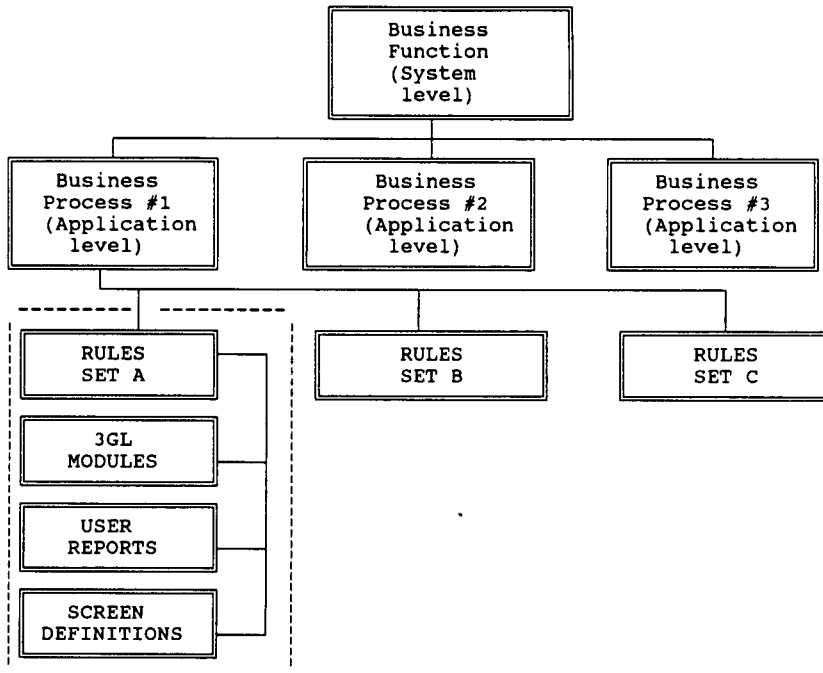
Figure 1.   Repository Objects in the Integrated CASE Environment

function points are used for measuring the functionality or size of a CASE-delivered system, any related software development effort estimates should be adjusted to reflect the functionality added by reused code that did not require commensurate effort.

Thus, although the five function types represent the intrinsic functionality of CASE-developed systems, it would be useful to have a mechanism that translates functionality into the more natural building blocks of modules and objects in an object-based CASE environment. In related research, we have investigated a solution to the problem of mapping between CASE objects and the function types [2]. The proposed mapping forms the basis of automating function points analysis in ICE. This could effectively circumvent the problems of effort, time, and inconsistency in manually counting the function points of CASE-delivered systems. However, estimation of function points remains inefficient and unintuitive in such CASE environments.

2.2. Step 2—Classification into Simple, Average, and Complex Types

Classification of the instances of the five function types into three levels of complexity is the second step in function point analysis. This procedure yields FUNCTION-COUNTS. The original complexity weights that distinguish the different complexity levels were determined by Albrecht in the 1970s by trial and error [1]. Symons [26] concluded that a new set of weights might need to be calibrated for any new technology, or new development environment. Clearly, CASE qualifies as a technology that differs from the traditional 3GL development activities for which Albrecht's weights were initially developed.

It is useful to keep in mind that the rationale for decomposing each function type into simple, average, and complex levels came from a realization that each represented a different level of functionality delivered to the user. For purposes of estimating costs, this translates into different amounts of time to code each complexity type. However, when CASE development techniques are used, the differential between the time required to code a simple type and a complex type may not be as large as in a 3GL development environment. The ability to do object-based development, to reuse code, and to generate code may contribute to an increased uniformity in the levels of effort required for developing different complexity types.

Our proposition, then, is that the complexity differentials in CASE FUNCTION-COUNTS may not lead to a significant improvement in estimating the actual development labor consumed. Thus, the complexity classification used in function points analysis may not only need recalibration, but, in fact, may not be worth the extra effort when little or no gain in estimation performance is made. In CASE environments that exhibit some of the characteristics of ICE, it may be worthwhile to consider a simpler, aggregate count for each of the five function types, without further classification by complexity level.

Other problems with the classification of function types into three levels of complexity include increased subjectivity and measurement effort. Level of experience in software programming—and by analogy, with CASE tools—affects an analyst's perception of the complexity of a function type [18]. The time and effort involved in achieving this subclassification through CASE-generated documentation further adds to the cost of counting function points.

2.3. Step 3—Adjusting FUNCTION-COUNTS by the TECHNICAL-COMPLEXITY-FACTOR

Symons [26] advocates a more open-ended approach to the specification of the factors affecting the environmental complexity of software development activities. Availability of CASE utilities such as automatic code generation, graphics generation, and screen painting may reduce the development labor required to implement applications that would score high in terms of the TECHNICAL-COMPLEXITY-FACTOR (TCF) score. (Refer to Appendix 1 for details about the components of TCF.)Moreover, in the integrated CASE environment we have been studying, reuse affects development effort far more than any other factor [4]. In short, not all of the fourteen factors on the list for traditional 3GL development may still be relevant in CASE-based system development.

As a result, TCF may not explain a significant portion of the variation in labor consumed in developing a CASE-based software application. So the time and effort spent in calculating TCF would not be of value. (Note in Appendix 1 that TCF can take on values in the range of 0.65 to 1.35, and thus can adjust the final number of function points no more than 35 percent.)

Thus, it is worthwhile to assess the predictive ability of TCF and its components. At the same time, the effect of software reusability needs to be considered in more detail for the measurement procedure to be appropriate for CASE.

2.4. RAW-FUNCTION-COUNTS:
A Proposed Short-form Variation of Function Points

Based on interviews with ICE software development managers and this critique, we propose RAW-FUNCTION-COUNTS, a short form variation of FUNCTION-POINTS as a candidate metric appropriate for measuring outputs from object-based CASE environments. This metric is defined as follows:

$$RAW\text{-}FUNCTION\text{-}COUNTS = \sum_{t=1}^{5} FUNCTION\text{-}TYPE\text{-}INSTANCES_t$$

where *FUNCTION-TYPE-INSTANCES* = total number of instances of function type t in an application; and t = function types (External Inputs, External Outputs, Queries, External Interfaces and Logical Internal Files).

Step 1 from the function points analysis procedure is retained in the calculation of the RAW-FUNCTION-COUNTS metric. However, once all of the instances of the five function types have been identified, a simple sum is computed. Note that, unlike function point analysis, RAW-FUNCTION-COUNTS are not separated into different complexity levels and weighted as in step 2, nor are they adjusted for external complexity as in step 3. We will soon present results to compare RAW-FUNCTION-COUNTS and FUNCTION-POINTS in terms of their ability to predict development labor for ICE projects. This comparison will provide justification for the proposed elimination of steps 2 and 3 from the function points procedure as a means of saving time and effort, without significant loss in the predictive power of cost estimation for ICE projects.

3. A New Approach to Output Size Measurement for CASE

THE DATA WE HAVE AVAILABLE ON NINETEEN INVESTMENT BANKING PROJECTS developed using ICE offers an interesting opportunity to test these metrics. Although we have indicated at the outset that object-based ICE may not be representative of all CASE tools available in the market today, nevertheless object-based and object-oriented development methods increasingly are utilized in CASE environments.[1] Further, they are widely believed to represent the standard analysis and design, and construction methodologies for software development in the 1990s [6, 8].

3.1. Object-Based Development in ICE

ICE applications are comprised of *objects* that act as building blocks, which the programmer uses to synthesize the functionality required by an application system. Objects provide specific, well-defined functionality in handy, ready-to-use chunks

of code. Definitions and code contents of all such objects are stored in the *central repository* that enforces standardization conventions regarding the definitions of objects of various types. An object need only be written once, and all subsequent applications that need to deliver similar functionality can make use of the relevant object from the repository. Thus, if a system needs to deliver functionality not already embodied in an existing object, a new object may be created according to the standard conventions for its definition. This discipline in the object storage and application version-management features of the central repository streamlines the process of creating software by enabling the reuse of existing objects.

The central repository stores information about the different kinds of objects used in applications developed with the tool. Examples of object types defined for the CASE tool we studied are: RULE SETS, 3GL MODULES, SCREEN DEFI-NITIONS, and USER REPORTS. Each object type is defined rigorously in order to make the process of software development conducive to object reuse. A RULE SET is a collection of instructions and routines written with the high level language of the CASE tool. A RULE SET would be thought of as "the program" if this were 3GL development. As such, RULE SETS offer the developer extensive flexibility in encoding functionality and generally allow all of the most commonly required functions to be constructed. For more complex or uncommon functions, a 3GL MODULE object can be constructed or reused. A 3GL MODULE most often represents a precompiled procedure, originally written using a third-generation language. A SCREEN DEFINITION is the logical representation of an on-screen image, and delivers to the user the functionality that would normally be associated with the user interface. Finally, a USER REPORT means much the same as it does in development environments other than ICE.

All objects associated with an application are functionally organized into an *object hierarchy*, which is stored in the central repository. An application consists exclusively of these objects and each application can be identified by a high-level BUSINESS PROCESS at the root of the hierarchy. A BUSINESS PROCESS calls other RULE SETS, which in turn use other RULE SETS or 3GL MODULES. These in turn can communicate with a SCREEN DEFINITION, or create a USER REPORT. (Figure 1 gives an idea of the relationships among the objects.)

The relationships among objects (which RULE uses which 3GL MODULE, which SCREEN invokes which VIEW, etc.) are themselves stored in the central repository. Collectively, the set of object instances and relationships between them make up the model of an application, and this model subsequently can be used by an analyst to identify the objects and the object instances that comprise an application. Identification of such objects has two important benefits. First, this process follows the natural building process of CASE systems, is intuitive, and therefore has the potential to be more accurate and consistent. Second, the representation of the application stored in the repository can be utilized to facilitate the automation of object identification. Such automation would lead to considerable savings in the effort and cost involved in collecting information about the objects used, and motivate implementation of revised procedures for CASE output measurement.

3.2. Object Point Analysis: A New Direction for Software Output Size Metrics

Do objects represent the functionality of an ICE application? Will knowing the number of objects that comprise a system provide sufficient information to estimate the labor required to build it? Is *object point analysis* a useful analogy for function point analysis in development environments such as ICE? We will argue that the size and functionality delivered by an ICE application can be derived from the aggregation of the objects used to build it.

To explore these questions further, we conducted two sets of interviews with managers and analysts experienced in the use of ICE within the organization. The first set involved Delphi sessions in which groups of four or five project managers were asked to estimate the time required to build a small application involving a wide variety of functionality requirements. The Delphi sessions were taped for later analysis. To expand on the themes that unified the approaches used for reaching group estimates of development labor, we conducted a second set of individual follow-up interviews. Project managers responsible for developing and estimating projects were asked more focused questions regarding how they would estimate development labor using ICE objects as the basis of their estimates.

The results of our Delphi sessions and individual interviews indicated that project managers employ estimation heuristics to assess the number of different types of objects that need to be developed for a project. Use of heuristics by experts for the estimation of software development costs has been reported previously in other development environments [27]. Using these heuristics, an ICE project manager initially estimates the number of RULE SETS, 3GL MODULES, SCREEN DEFINI-TIONS, and USER REPORTS that he or she believes will comprise the application software that is to be delivered.

However, similar to the function types in function points, different objects exhibit different levels of complexity and functionality, and also require different amounts of development labor to construct. The project managers we interviewed classified occurrences of object types into three levels of complexity. Each complexity level within an object type was regarded as requiring a different number of days to develop. Project managers' object-effort estimates are summarized in Table 3 in terms of the *average time required to build a given object type*.

We utilize the means of their effort estimates for the object complexity levels because we have not yet fully explored the set of dimensions upon which the managers classify objects into complexity levels. A deeper investigation into the nature of heuristics for estimation and classification of objects in ICE environments is required in order to specify dimensions of object complexity. We will then be in a position to generate object-effort tables from a database of actual projects developed with ICE.

Two new object-based output measures are suggested by our analysis. The first, OBJECT-COUNTS, is determined by summing the instances of individual objects of the four types. The second, OBJECT-POINTS, is determined by weighting each object type by the development effort associated with it, as shown in Table 3. OBJECT-COUNTS and OBJECT-POINTS are defined more formally below:

Table 3   Project Manager Development Effort Heuristics

OBJECT TYPE	PROJECT MANAGER EFFORT HEURISTICS (AVERAGE)
RULE SETS	3 days
3GL MODULES	10 days
SCREEN DEFINITIONS	2 days
USER REPORTS	5 days

$$OBJECT\text{-}COUNTS = \sum_{t=1}^{4} OBJECT\text{-}INSTANCE_t$$

$$OBJECT\text{-}POINTS = \sum_{t=1}^{4} OBJECT\text{-}INSTANCE_t * OBJECT\text{-}EFFORT\text{-}WEIGHT_t$$

where $OBJECT\text{-}INSTANCE_t$ = total number of instances of object type t in an ICE application; $OBJECT\text{-}EFFORT\text{-}WEIGHT_t$ = average development effort associated with object type t, based on project manager heuristics; and t = object type (RULE SET, 3GL MODULE, SCREEN DEFINITION and USER REPORT).

Hereafter, we will refer to OBJECT-COUNTS and OBJECT-POINTS as object points analysis metrics.

4. Evaluation of Alternate Metrics for Measuring CASE Outputs

WE NEXT COMPARE THE OBJECT POINTS ANALYSIS METRICS, OBJECT-COUNTS and OBJECT-POINTS, with the function point analysis-related metrics, RAW-FUNC-TION-COUNTS and FUNCTION-POINTS, as candidates for the measurement of outputs in object-based CASE development.

4.1. Modeling Output Metric Performance

To test the performance of the four metrics for estimation of software development labor, we estimated a set of regression models of similar functional form to predict *development effort* in terms of each of the output metrics. The regression results can be used to indicate the extent to which a given output metric is able to explain the variance in development effort, after it has been adjusted to reflect the beneficial leverage on productivity created by including reused code. When high levels of reuse are observed, the resulting functionality of a system alone will not be a very good predictor of the labor required to build it: reused code does not require an equivalent amount of labor input to construct and to implement.

For the functionality embodied in the reused code to be reflected in the development labor logged against the project, we adjusted PERSON-MONTHS of effort by a factor that represents a rough estimate of the leverage on development productivity provided by reused code [2]. This adjustment can be effected using a metric called REUSE-LE-VERAGE. This measure for reuse is based on a second metric called *NEW-OBJECT-PERCENT*, that we have proposed in related CASE productivity research [4]. NEW-OBJECT-PERCENT is meant to provide a reading on the portion of an application that must be built from scratch in a CASE environment that supports reusability. As the value for NEW-OBJECT-PERCENT approaches 100 percent, no leverage is created. Formal definitions for NEW-OBJECT-PERCENT and REUSE-LEVERAGE are given below:

$$NEW-OBJECT-PCT = \frac{NUMBER-UNIQUE-OBJECTS-BUILT-FOR-APPLICATION}{TOTAL-NUMBER-OBJECTS-COMPRISING-APPLICATION},$$

$$REUSE-LEVERAGE = \frac{1}{NEW-OBJECT-PCT}.$$

Our metrics for measuring reuse agree with the reuse measurement approaches advocated by Neighbors [20] for 3GL environments. REUSE-LEVERAGE, as the inverse of NEW-OBJECT-PERCENT, measures the average number of times application objects are reused within an application [2]. This is a *leverage metric*, which means that the functionality delivered through reusable software would add to the required labor estimates, if it were necessary to built the same functionality from scratch. Of course, in CASE development with opportunities to reuse existing software, this additional labor need never be expended. Thus, in order to use existing metrics that capture the functionality of the outputs of software development to predict the labor needed, we adjust PERSON-MONTHS expended by multiplying it by REUSE-LEVERAGE.

The estimation model we used to compare the various output metrics has the following mathematical form:

$$PERSON-MONTHS * REUSE-LEVERAGE =$$
$$\beta_0 + (\beta_1 * OUTPUT-METRIC * DUMMY1) +$$
$$(\beta_2 * OUTPUT-METRIC * DUMMY2) + \varepsilon$$

where *PERSON-MONTHS* = number of person months of development labor consumed in constructing the project; *REUSE-LEVERAGE* = total number of objects used in an application divided by the number of unique objects used in the application; *OUTPUT-METRIC* = application output, as measured by FUNCTION-POINTS, RAW-FUNCTION-COUNTS OBJECT-COUNTS or OBJECT-POINTS; *DUMMY1* = 1 if project was constructed in Year 1, and 0 otherwise; *DUMMY2* = 1 if project was constructed in Year 2, and 0 otherwise; β_0, β_1, β_2 = parameters to be estimated in the regression; and ε = a normally distributed error term.

A model incorporating the binary variables *DUMMY1* and *DUMMY2* enables us to represent information about the relative productivity of the twelve projects constructed

Table 4   Data for Four Software Development Output Measurements

ICE PROJ. NO.	YEAR 1/ YEAR 2 PROJECT?	NEW- OBJECT- PERCENT	RAW- FUNCTION- COUNTS	FUNCTION- POINTS	OBJECT- COUNTS	OBJECT- POINTS
1	YEAR 1	23.2%	522	2250.08	202	1768
2	YEAR 1	100.0%	51	170.56	27	144
3	YEAR 1	54.3%	82	300.14	59	499
4	YEAR 1	35.1%	64	264.60	74	600
5	YEAR 1	61.0%	286	1273.70	46	271
6	YEAR 1	49.3%	108	352.50	69	523
7	YEAR 1	48.1%	65	494.08	77	231
8	YEAR 1	93.1%	24	97.92	29	87
9	YEAR 1	96.2%	27	148.41	27	123
10	YEAR 1	69.0%	83	385.14	71	376
11	YEAR 1	44.8%	234	1092.00	29	124
12	YEAR 1	45.7%	46	241.82	57	276
13	YEAR 2	26.6%	559	3812.40	368	2258
14	YEAR 2	34.7%	372	1772.40	187	1262
15	YEAR 2	29.2%	587	3475.20	335	2023
16	YEAR 2	78.1%	28	135.00	23	163
17	YEAR 2	23.1%	865	5876.25	478	2698
18	YEAR 2	16.1%	608	3712.80	619	3657
19	YEAR 2	32.8%	194	886.58	259	1915
MEAN	--	50.5%	252.89	1407.45	159.79	999.89
MAX	--	100.0%	865	5876.25	619	3657
MIN	--	16.1%	24	97.92	23	87
STD DEV	--	26.0%	256.61	1665.14	174.92	1068.38
MEAN/ STD DEV	--	1.94	0.986	0.845	0.913	0.936
COUNT	YEAR 1:12 YEAR 2: 7	--	--	--	--	--

in Year 1, when the CASE tool was itself under construction, and the seven projects developed later in Year 2. Year 2 projects tended to be much larger development efforts, where the power of a more well-developed CASE development tool set was evident and higher levels of reuse were observed. As a result, each of the years of ICE project development exhibited different productivity levels. Our study of Year 2 projects indicated a substantial gain in productivity when compared to Year 1 projects [4]. Clearly, developers' use of the tool had begun to mature by Year 2. The model specified above accounts for this difference in development productivity over the two years.

4.2. Estimation Performance Results

Data for the measures used to estimate the regression models discussed above are presented in Table 4. Our first step was to examine correlations between the output metrics. Table 5 presents the correlation results.

The correlation between RAW-FUNCTION-COUNTS and FUNCTION-POINTS was 0.981, while the correlations between the function point metrics and the object-

Table 5   Output Metrics Correlation Matrix

OUTPUT METRICS	CORRELATIONS			
	RAW-FUNCTION-COUNTS	FUNCTION-POINTS	OBJECT-COUNTS	OBJECT-POINTS
RAW-FUNCTION-COUNTS	-	0.981	0.871	0.868
FUNCTION-POINTS	0.981	-	0.889	0.862
OBJECT-COUNTS	0.871	0.889	-	0.986
OBJECT-POINTS	0.868	0.862	0.986	-

based metrics, OBJECT-COUNTS and OBJECT-POINTS, were somewhat lower (0.889 maximum). Since function point analysis is well-established and well-validated, correlations between FUNCTION-POINTS and the object-based metrics are an indication of the *convergent validity* of the proposed metrics. Low correlations, on the other hand, could mean that the proposed metrics are not good measures of the construct that function points purports to measure, namely, application functionality that is delivered to the user. Alternatively, low correlations could indicate that the proposed object-based metrics complement FUNCTION-POINTS by measuring a construct that is ignored by FUNCTION-POINTS.

For our data set, the easier to collect RAW-FUNCTION-COUNT metric potentially could be as useful a measure of output as FUNCTION-POINTS, if estimation accuracy can be maintained. The same, however, may not be true for the object point analysis metrics. OBJECT-COUNTS and OBJECT-POINTS may measure a different aspect of the applications' functionality, or an entirely different characteristic of the output that we have not yet identified.

Our next step was to examine the quality of the development effort estimates produced by the metrics. Regression results for the four estimation models discussed above are presented in Table 6, including information about the estimated parameters and the fit of the models in terms of R^2.

The RAW-FUNCTION-COUNTS and FUNCTION-POINTS metrics were estimated to explain about 76 percent and 75 percent of the variance in PERSON-MONTHS * REUSE, respectively. The similarity between the two metrics' estimation performance is readily explained. Projects in the data set exhibited relatively similar values for the TECHNICAL-COMPLEXITY-FACTOR representing the complexity of the implementation environment: this did not vary much across the applications. Thus, the results support the proposition that estimating complexity differentials for ICE-delivered FUNCTION-COUNTS may not lead to substantial improvement in estimating development labor.

OBJECT-COUNTS demonstrated a similar performance in estimating PERSON-MONTHS * REUSE. R^2 for the estimation model involving OBJECT-COUNTS fell

Table 6   Results for Estimation Performance of Metrics

SOFTWARE OUTPUT MEASUREMENT METRIC	COEFFICIENT ESTIMATES (SIGNIFICANCE LEVELS)			REPORTED VALUE OF R-SQUARED
	$\beta 0$	$\beta 1$	$\beta 2$	
RAW-FUNCTION-COUNTS	13.14 (.26)	0.39 (.001)	0.16 (.001)	.76
FUNCTION-POINTS	14.88 (.20)	0.09 (.001)	0.02 (.001)	.75
OBJECT-COUNTS	-2.80 (.85)	1.13 (.001)	0.25 (.001)	.70
OBJECT-POINTS	12.63 (.31)	0.13 (.001)	0.04 (.001)	.73

to 70 percent, a 6.7 percent decrease from FUNCTION-POINTS. The OBJECT-POINTS metric performed marginally better than OBJECT-COUNTS; it demonstrated the ability to explain 73 percent of the variance in the output metric, approximating the performance of FUNCTION-POINTS. Once again, the regression results indicate the goodness of fit of the model, and thereby provide evidence in support of the *estimation performance* of the metrics. These results, however, are inconclusive insofar as whether one metric is better than the other as a measure of the intrinsic size and functionality of the CASE-developed software. More data are required to answer this question.

To give the reader a sense of the estimation qualities of each of the four metrics, estimates for development labor for the nineteen projects were calculated based on the coefficients obtained from the regression models. These estimates are shown in Table 7, along with the actual values of PERSON-MONTHS of development labor.

The third category of results is derived from an interpretation of the parameter estimates (β_0, β_1, and β_2). The majority of the parameters obtained from these models were positive and significantly different from zero. A side result of the modeling approach we have used is that it provides information on the productivity gain ratios between Year 1 and Year 2 development, based on the estimated parameters from the regressions. Table 8 presents the *productivity gain ratios*, β_1/β_2, for each of the output metrics estimations.

Although the Year 1 to Year 2 productivity gain ratios exhibit considerable variance, they generally demonstrate the extent to which productivity increased in the firm's use of CASE over the two years. The low end of the range of productivity gain ratios is about 2.44 times for RAW-FUNCTION-COUNTS. Using FUNCTION-POINTS and OBJECT-COUNTS as estimators led to the largest estimated productivity ratios between the years. One possible interpretation of these differences is that RAW-FUNCTION-COUNTS underestimates output because it treats the labor requirements of different complexity levels uniformly. However, as the functionality and complexity embodied in Year 2 projects increased, underestimation of output by RAW-FUNC-TION-COUNTS increased more than proportionately. As a result, the productivity gain ratio estimated by RAW-FUNCTION-COUNTS were the least. FUNCTION-

Table 7   Actual and Estimated Labor Using Four Output Metrics*

| PROJECT NUMBER | ACTUAL PERSON MONTHS LABOR | ESTIMATED-PERSON-MONTHS OF LABOR | | | |
		RAW- FUNCTION- COUNTS	FUNCTION- POINTS	OBJECT- COUNTS	OBJECT- POINTS
1	59.33	50.84	49.23	52.44	56.00
2	40.94	33.27	29.84	27.79	31.26
3	27.33	24.71	22.37	34.77	41.92
4	28.89	13.48	13.37	28.44	31.68
5	71.89	76.87	77.21	30.08	29.10
6	16.39	27.49	22.58	37.16	39.59
7	26.17	18.66	28.00	40.61	20.45
8	7.56	21.06	21.85	27.98	22.24
9	23.67	22.85	26.78	26.68	27.41
10	47.89	31.67	33.57	53.57	42.29
11	8.17	47.26	49.57	13.46	12.85
12	12.78	14.30	16.49	28.23	22.09
13	38.11	27.13	29.00	23.97	24.80
14	20.89	25.07	20.35	15.41	20.01
15	26.06	31.08	29.41	23.88	24.77
16	4.72	13.74	14.23	2.35	14.41
17	36.00	34.79	36.96	27.23	25.16
18	12.94	17.67	17.16	24.71	23.05
19	23.11	14.42	12.06	20.53	26.56

* In each case above, the estimated coefficients presented in Table 6 were applied to the output metric data from Table 4 using the following model:

$$\text{PERSON-MONTHS} * \text{REUSE-LEVERAGE} = b_0 + (\beta_1 * \text{OUTPUT-METRIC} * \text{DUMMY1})$$
$$+ (\beta_2 * \text{OUTPUT-METRIC} * \text{DUMMY2}) + \varepsilon.$$

Since the application of the estimated coefficients to the data yields (PERSON-MONTHS * REUSE-LEVERAGE), the final step is to divide by the reuse measure, REUSE-LEVERAGE, to arrive at estimated PERSON-MONTHS of software development labor.

POINTS, while accurately capturing the higher functionality of the more complex applications developed in Year 2, may tend to overstate the labor required to create them. The mean of the productivity gain ratio corresponding to the four metrics was 3.68, and this was most closely matched by the productivity gain ratio of OBJECT-POINTS at 3.25. Thus, each of the models provides clear evidence for the extent of productivity gains observed as use of the CASE tool matured in the firm.

5. Concluding Remarks

OUR INVESTIGATION INTO THE PERFORMANCE of two function point analysis-related metrics and two object-based metrics suggests that there may exist viable alternate approaches for measuring the outputs of the CASE-development process. This study was conducted as an exploratory investigation to provide us with a basis for further developing measurement approaches for object-based CASE environments. The reader should bear in mind that conclusions regarding the performance of the alterna-

Table 8   Year 1 and Year 2 Productivity Ratios Based on Estimated Parameters

SOFTWARE OUTPUT MEASUREMENT METRIC	PRODUCTIVITY GAIN RATIO: YEAR 1 VERSUS YEAR 2 (BASED ON β PARAMETERS)
RAW-FUNCTION-COUNTS	0.39/0.16 = 2.44
FUNCTION-POINTS	0.09/0.02 = 4.50
OBJECT-COUNTS	1.13/0.25 = 4.52
OBJECT-POINTS	0.13/0.04 = 3.25

Note: The productivity gain ration is computed as β_1/β_2. The values presented in the table should be interpreted as the ratio of additional labor required in Year 1 compared to what was required in Year 2.

tive metrics that we examined in this study were obtained based on nineteen software development projects drawn from a single organization. As such, our findings should be interpreted within the limited validity of the study.

5.1. Contributions

The results of this study are summarized in Table 9.

Two alternative measurement approaches exhibited strong potential for further development and validation in object-based CASE environments. The RAW-FUNC-TION-COUNTS metric proved to be comparable to FUNCTION-POINTS in terms of its estimation performance, and it is readily implemented with less effort and at a lower cost. We also achieved considerable success in our test of the OBJECT-COUNTS and OBJECT-POINTS metrics as estimators for software development labor. Both approach the estimation capabilities exhibited by FUNCTION-POINTS, although they better match the ways that ICE developers think about the software they build. Our approach to estimating the productivity gain ratios from the use of CASE in Year 2 versus Year 1 also has important managerial implications for research on CASE productivity. Although the number of data points available for our analysis was small, the lessons and insights obtained by studying them carefully can help to build a broader base of results and experience in the area of CASE productivity assessment. Combining the results of this research with results we have obtained in related work provides considerable evidence to suggest that the use and availability of key development facilities made available with the CASE tool affect productivity. This effect is most likely enhanced by the wider range of opportunities for reuse and a development environment that is more stable and better understood by developers.

At this stage, we have no information about whether the results would also hold true in other integrated CASE development environments. This research question can only be

Table 9   Study Findings: Four Output Metrics for CASE Output Measurement

METRIC		OVERALL PERFORMANCE OF THE PROPOSED METRIC
RAW- FUNCTION- COUNTS	*	Performed approximately as well as FUNCTION-POINTS in estimating development labor, explaining 76% of the variance.
	*	Tends to understate the productivity gain ratio for Year 2 to Year 1 projects at 2.44 times; probably due to lack of treatment of complexity of functions counted.
	*	Relatively high correlation between RAW-FUNCTION-COUNTS and FUNCTION-POINTS possible since TECHNICAL-COMPLEXITY-FACTORS for ICE project did not vary much.
	*	Given the small difference in estimation performance, RAW-FUNCTION-COUNTS should be a candidate short-form metric for use with ICE projects.
FUNCTION- POINTS	*	Base case metric for this research explained 75% of variance in development labor.
	*	Year 2 to Year 1 productivity gain ratio estimated at 4.50 times.
	*	Thus, FUNCTION-POINTS provides a good, though costly and labor-intensive metric to collect.
OBJECT- COUNTS	*	OBJECT-COUNTS performed the poorest of the four metrics, explaining only 70% of the variance of development labor.
	*	Project manager interviews and Delphi estimation sessions suggested that it makes sense to weight objects by object type to capture construction complexity.
	*	Provided high end estimate of Year 2 to Year 1 productivity gain ratio at 4.52 times, similar to FUNCTION-POINTS.
	*	Given the ease and low cost of use, OBJECT-COUNTS appears to provide a useful metric for ICE software development project management.
OBJECT- POINTS	*	OBJECT-POINTS appear to provide a slight improvement over OBJECT-COUNTS in estimation performance; the former metric explained 73% of the variance of development labor.
	*	OBJECT-POINTS most closely matches the mean of the four Year 2 to Year 1 productivity gain rations, 3.68, at 3.25 times.

investigated using data sets that involve multiple organizations. However, we believe that some of the characteristics of the development environment we studied and utilized in testing the metrics are present in other object-based CASE environments. These include the use of objects that represent application functionality, opportunities to reuse code and a centralized repository. If these are available to developers, then there is a reasonable likelihood that our results will generalize to other CASE environments.

5.2. Future Research

In future research, we intend to further develop object points analysis as an output size measurement approach that is tailored to and built into the object-based CASE development process. The first step we will take is to examine another more detailed

object-based metric in which each object is weighted by the approximate time it takes to construct. Our Delphi sessions and individual project manager interviews suggested that project managers may further distinguish among the complexity levels of the various objects that they build into ICE applications (similar to the function type complexity levels of function point analysis). Additional work needs to be done to identify the dimensions of the objects that define their complexity levels. Our intent is to extend the analogy between function point analysis and object point analysis based on empirical evidence. Perhaps this line of investigation will also enable us to determine the set of circumstances under which metrics that capture application functionality perform better as cost estimators than metrics that identify application objects. To reach this research goal, we hope to study a larger set of projects within the same organization and to extend our analyses to the projects of other organizations that have implemented object-based ICE.

Another open question is the automation of object points analysis. Object points analysis reporting tools should be deployed to analyze the changing contents of the repository as an application is constructed. Since objects were found to match project managers' mental model of the functionality of software developed with object-based CASE, information about objects would be useful to promote improved software development process control. The measurement of OBJECT-POINTS, and another variation that involves weights for the complexity of objects that make up the application, can be automated at low cost, once we have solved the problem of dimensioning object complexity. (See [2] for a discussion of how function point analysis and code reuse analysis can be automated within ICE.)

When senior managers of software development operations have such tools available, the stage is set for new approaches to the management of software development activities—process management and process control [9]. To date, the process of tracking software development operations has largely been based on single point estimates of cost and productivity, for example, made at the inception and completion of a project. But, the data made available by automating the measurement process as a project proceeds through the development life cycle offer many possibilities for rich and insightful analyses that cannot be conducted using traditional performance tracking approaches. Management can increase its effectiveness by proactively fine-tuning the process of software development at the project level as it occurs, rather than adjusting it for future development.

REFERENCES

1. Albrecht, A.J., and Gaffney, J.E. Software function, source lines of code, and development effort prediction: a software science validation. *IEEE Transactions on Software Engineering*, 9, 6 (November 1983), 639–647.

2. Banker, R.D.; Fisher, E.; Kauffman, R.J.; Wright, C.; and Zweig, D. Automating software development productivity metrics. Working Paper, Center for Research on Information Systems, Stern School of Business, New York University, October 1990.

3. Banker, R.D., and Kauffman, R.J. Automated software metrics, repository evaluation and software asset management: new perspectives for computer-aided software engineering environments. Working Paper, Center for Research on Information Systems, Stern School of Business, New York University, March 1991.

4. Banker, R.D., and Kauffman, R.J. Reuse and functionality: an empirical study of integrated computer-aided software engineering (ICASE) technology at the First Boston Corporation. *MIS Quarterly*, September 1991.

5. Booch, G. What is and what isn't object-oriented design. *Ed Yourdon's Software Journal*, 2, 7–8 (Summer 1989), 14–21.

6. Bouldin, B.M. CASE: measuring productivity—what are you measuring? why are you measuring it? *Software Magazine*, 9, 10 (August 1989), 30–39.

7. Dreger, J.B. *Function Point Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1989.

8. Goldstein, D.G. Object oriented programming. *DEC Professional*, 9, 2 (February 1990).

9. Humphrey, W.S. *Managing the Software Process*. Reading, MA: Addison-Wesley, 1990.

10. *Application Development Productivity Strategy*, World-wide IBM User Group, Application Development Joint Project, June 1989.

11. *Proceedings of the International Function Points Users Group*, International Function Points Users Group, 1988.

12. Jones, T.C. *Programming Productivity*. New York: McGraw-Hill, 1986.

13. Jones, T.C. New look at languages. *ComputerWorld*, November 1988.

14. Kemerer, C.F. An empirical validation of software cost estimation models. *Communications of the ACM*, 30, 5 (May 1987), 416–429.

15. Kemerer, C.F. An agenda for research in the managerial evaluation of computer-aided software engineering (CASE) tool impacts. *Proceedings of the 22nd Hawaii International Conference on Systems Sciences*, Hawaii, IEEE, January 1989.

16. Kemerer, C.F. Reliability of function points measurement: a field experiment. Working Paper, Sloan School of Management, MIT, December 1990.

17. Keyes, J. Peeling back layers of the quality equation. *Software Magazine*, 11, 6 (May 1991), 43–61.

18. Low, G.C., and Jeffrey, D.R. Function points in the estimation and evaluation of the software process. *IEEE Transactions on Software Engineering*, 16, 1 (January 1990), 64–71.

19. Meyer, B. *Object-Oriented Software Construction*. Englewood Cliffs, NJ: Prentice-Hall, 1988.

20. Neighbors, J.M. The DRACO approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, SE–10, 5 (September 1984), 564–574.

21. Norman, R.J., and Nunamaker, J.F., Jr. CASE productivity perceptions of software engineering professionals. *Communications of the ACM*, 32, 9 (September 1989), 1102–1108.

22. Nunamaker, J.F., Jr., and Chen, M. Software productivity: a framework of study and an approach to reusable components. *Proceedings of the 22nd Hawaii International Conference on System Sciences*, Hawaii, IEEE, January 1989, pp. 957–958.

23. Rubin, H.A. Macroestimation of software development parameters: the ESTIMACS system. *IEEE Softfair Conference on Software Development Tools, Techniques and Alternatives*, 1983.

24. Rudolph, E.E. Evaluation of a fourth generation language. *Proceedings of ACS and IFIP Joint Symposium on Information Systems*, April 1984, pp. 148–165.

25. Senn, J.A., and Wynekoop, J.L. Computer aided software engineering (CASE) in perspective. Working Paper, Information Technology Management Center, College of Business Administration, Georgia State University, 1990.

26. Symons, C.R. Function point analysis: difficulties and improvements. *IEEE Transactions on Software Engineering*, 14, 1 (January 1988), 2–10.

27. Vicinanza, S.; Mukhopadhyay, T.; and Prietula, M.J. Software effort estimation: a study of expert performance. Working Paper 89–002, Center for the Management of Technology, Graduate School of Industrial Administration, Carnegie Mellon University.

APPENDIX 1. The Function Point Analysis Procedure

STEP 1: Identification of Function Types
Identify each functionality unit and classify it into five user function types:

• *External Outputs* are items of business information processed by the computer for the end user.

• *External Inputs* are data items sent by the user to the computer for processing, or to make additions, changes or deletions.
• *Queries* are simple outputs; they are direct inquiries into a database or master file that look for specific data, use simple keys, require immediate response, and perform no update functions.
• *Logical Internal Files* are data stored for an application, as logically viewed by the user.
• *External Interface Files* are data stored elsewhere by another application, but used by the one under evaluation.

This step yields a count for each of the five different function types. (For the purposes of this research, we refer to the sum of the count of the five function types as RAW-FUNCTION-COUNTS (RFC). This metric is never calculated within the function points analysis procedure.)

STEP 2: Classification of Simple, Average, and Complex Function Types
The individual counts by function type are further classified into three complexity levels (simple, average, complex), depending on the number of data elements contained in each function type instance and the number of files referenced. Each function complexity subtype is weighted with numbers reflecting the relative effort required to construct the function. For example, according to Albrecht's weighting scheme, a simple input type would be weighted by 3, while a complex input type would be weighted by 4. Additional details about the FUNCTION-COMPLEXITY-SCORES are shown below:

FUNCTION TYPE (f)	FUNCTION-COMPLEXITY-SCORES (c)		
	Simple	Average	Complex
Inputs	3	4	6
Outputs	4	5	7
Interfaces	5	7	10
Queries	3	4	6
Files	7	10	15

FUNCTION-COUNTS (FC) summarizes the weighted counts for the five function types as follows:

$$\sum_{t=1}^{5} \sum_{c=1}^{3} FUNCTION\text{-}TYPE_t * FUNCTION\text{-}COMPLEXITY\text{-}SCORE_c$$

STEP 3: Adjusting FUNCTION-COUNTS by
TECHNICAL-COMPLEXITY-FACTOR
The adjustment factor reflects application and environmental complexity, expressed as the degree of influence of fourteen "application characteristics" (f). Each characteristic is rated on a scale of 0 to 5 (COMPLEXITY-FACTOR-VALUE), and then all scores are summed. The TECHNICAL-COMPLEXITY-FACTOR (TCF) = 0.65 + (0.01 * $\Sigma_{f=1-14}$ COMPLEXITY-FACTOR-VALUE$_f$). The fourteen factors are shown below:

1. Data Communications	8. On-line Update
2. Distributed Functions	9. Complex Processing
3. Performance	10. Re-Usability
4. Heavily-used Config.	11. Installation Ease
5. Transaction Rate	12. Operational Ease
6. On-line Data Entry	13. Multiple Sites
7. End-User Efficiency	14. Facilitate Change

Finally, FUNCTION-POINTS (FP) are calculated as FC * TCF.