THE BUSINESS CASE FOR AUTOMATING SOFTWARE METRICS IN OBJECT-ORIENTED COMPUTER AIDED SOFTWARE ENGINEERING ENVIRONMENTS

by

Rajiv D. Banker

Robert J. Kauffman

THE BUSINESS CASE FOR AUTOMATING SOFTWARE METRICS IN OBJECT-ORIENTED COMPUTER AIDED SOFTWARE ENGINEERING ENVIRONMENTS

by

Rajiv D. Banker
Arthur Andersen Professor of Accounting and Information Systems
Carlson School of Business
University of Minnesota
Minneapolis, MN 55455

and

Robert J. Kauffman
Assistant Professor of Information Systems
Leonard N. Stern School of Business
New York University
New York, NY 10006

June 1990

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN IS-90-14

We wish to acknowledge Mark Baric, Gene Bedell, Tom Lewis and Vivek Wadhwa for the access they provided us to data on software development projects and managers' time throughout our field study of CASE development at the First Boston Corporation and Seer Technologies. We also wish to thank Eric Fisher, Rachna Kumar, Charles Wright and Dani Zweig for their contributions to various aspects of the work reported on in this paper. Finally, we thank the National Science Foundation for partial funding of the data collection under grant No. SES-8709044. All errors in this paper are the responsibility of the authors.

Presented at the *Workshop on IT Strategy and Architecture*, Wharton School of Business, University of Pennsylvania, Philadelphia, PA, June 22, 1990.

THE BUSINESS CASE FOR AUTOMATING SOFTWARE METRICS IN OBJECT-ORIENTED COMPUTER AIDED SOFTWARE ENGINEERING ENVIRONMENTS

Abstract

This paper makes the business case for automating the collection of software metrics for gauging development performance in integrated computer aided software engineering (**CASE**) environments that are characterized by an object-oriented development methodology and a centralized repository. The automation of function point analysis is discussed in the context of such an integrated CASE environment (**ICE**). We also discuss new metrics that describe three different dimensions of code reuse -- leverage, value and classification -- and examine the possibility of utilizing objects as means to estimate software development labor and measure productivity. We argue that the automated collection of these software metrics opens up new avenues for refining the management of software development projects and controlling stra_egic costs.

1. INTRODUCTION

As the 1990s begin, large-scale investments in computer aided software engineering (**CASE**) are becoming increasingly common as firms seek new ways to deal with the problem of managing the strategic costs of software development. But such investments also raise many questions for management (BOUL89, SENN90). For example, what are the features of a CASE tool that enable a firm to maximize development productivity, while maintaining acceptable quality and functionality? How does a CASE tool affect the activities associated with different portions of the software development life cycle? Are the benefits balanced, or are they concentrated in analysis and design rather than construction and testing? And, are the benefits of CASE sufficient to justify the high costs of implementing it? Is the move to modular, object-oriented software paying off? (POLL90)

The only way to obtain answers to these and other questions about the performance of investments in CASE is to develop measurement methods and programs that are well-suited to the new, emerging environments for software development. In this paper, we will discuss a research effort currently underway that is concerned with improving management's understanding of the potential of CASE and creating new approaches to measuring software development performance.

1.1. Investing in ICE -- An Integrated CASE Environment

A large investment bank located in New York City made the initial commitment to design and develop an *object-oriented, repository-based Integrated CASE Environment* (**ICE**) at a cost of tens of millions of dollars over the course of three years. ICE was built by the firm as a response to the problems it faced in developing and maintaining technically complex systems. The firm's computer operations are geographically distributed, and are required to perform effectively on a 24-hour basis.

Similar to its competitors in the investment banking industry, the firm had been experiencing rapidly mounting software costs that were expected to skyrocket as its trading activities expanded to provide global coverage. To achieve competitive performance in this environment previously required the firm's developers to program applications which ran on each of three hardware platforms (mainframe, minicomputer and microcomputer) in a different language -- COBOL, PL/I and C++, respectively. A CASE tool was needed that would support the programming of systems running simultaneously on all three platforms, and reduce the firm's reliance on three separate sets of highly skilled and costly programmers.

ICE systems are written in an *object-oriented* language which buffers programmers from the complexity of the firms's operating environment. Applications are later compiled in the appropriate languages for the relevant hardware platforms, and communications protocols for cooperative processing across platforms are handled without programmer intervention. The organization of the code into objects tends to be functional, and the various software functions can be allocated across hardware platforms in the most appropriate manner. (For an introduction to object-oriented software development methodologies and modular software, see GOLD89, MENG90, MEYE87, MEYE88 and POLL90.)

A special feature of ICE is its *object repository* (FISH90, HAZZ89). This includes all the definitions of the data and objects that make up the organization's business, and also all the pieces of software that comprise its systems. The motivation for having a single repository for all such objects is similar to that for having a single database for all data: a program, or a procedure, or a screen, or a report need only written once, no matter how many times it is used. Such reuse has the potential to decrease software development costs, and it forces the firm to more carefully engineer an information and information systems architecture which will form a solid base for the firm's business.

1.2. Software Metrics for Integrated CASE

Our research on software development productivity in this environment has led to a number of interesting discoveries. *First*, we have found that obtaining metrics that are traditionally used to gauge software development productivity in 3GL environments remains a very costly process when we translate them for use in the world of ICE development, despite the much improved quality of the documentation stored in the repository. More importantly, however, we have found that the amount of code reused in an application often represents a significant portion of its functionality, and that traditional metrics fail to take this into account. This has led us to develop models for development productivity in which traditional development productivity measures are adjusted to consider the level of reuse (BANK90A).

Second, during the course of our work we have also learned that there is very little research that has addressed the question of *how* reuse should be measured, especially when it is to be incorporated in a development productivity assessment model or methodology. We believe that this is less an oversight of prior research than a reflection of the realities of 3GL software development. In these development environments, modules of code may be reused or revised in ways that lead to new functionality. But, there are few tools to help a developer identify opportunities for reuse, and since the reused code is not stored in a central repository, it becomes very difficult to identify reuse other than with extensive manual effort. The primary result then has simply been not to measure it (BANK90D).

Third, we have learned that the functional organization of an ICE application into objects makes it practical to automate the analysis of code for the computation of a range of software development performance indicators, including metrics for productivity and complexity. The central repository also makes the automation of code reuse measurement practical, since it maintains a record of each object and where it has been used or reused (BANK90C).

Fourth, in our interviews and discussions with ICE developers we also have learned that the new development environment offers the possibility of utilizing new approaches to estimate the labor associated with the development of an ICE application. For example, function point analysis (which we will discuss in more detail shortly) is traditionally used to estimate development labor and to measure the resulting productivity of a development effort. However, we have learned that more intuitive and simplified metrics may serve equally well for ICE applications (BANK90B).

1.3. Outline of the Paper

In the remainder of the paper, we develop these ideas further. For example, Section 2 expands our critique of software metrics collection and the difficulties that managers will face in trying to make them work in CASE environments. Section 3 presents our proposals for new metrics which are tailored for use in software development environments that share some of the features of ICE. Section 4 discusses the business case for automating these metrics, and provides an overview of how it can be accomplished in an object-oriented, repository-based CASE environment. We argue that the automated collection of these software metrics opens up new avenues for refining the management of software development projects.

2. CRITIQUE: SOFTWARE METRICS FOR INTEGRATED CASE

2.1. Measuring Function Points

The magnitude of a software development effort depends upon several factors, including the amount of information processing accomplished by the system, the quality and the extent of the input and output interfaces provided to meet the users' needs, and environmental factors ranging from the quality of the hardware used by the programmers to the sophistication of the users requesting the software (SYMO88). Allan Albrecht of IBM originally proposed *function points* as a metric to capture the size of an application, so that software development activities could be evaluated for the outputs they create, and so that software development managers would have a tool to estimate the resources required to build systems of various sizes (ALBR79, ALBR83).

Function points are meant to provide a language-independent and implementation-independent measure of the functionality actually produced and delivered to the user. They differ from output measures (such as those based on source lines of code) that may reward verbose programming practices (KEME89). Since its introduction in the late 1970s, function point analysis has evolved, with the help of the International Function Point Users Group, into a well-accepted and operationally well-defined methodology (DREG89, ZWAN84).

Function points are computed by measuring the degree of functionality actually delivered to the user of the system, in terms of reports, inquiry screens, and so on. This functionality is determined by the number and complexity of inputs, outputs, internal files, external interfaces and queries that comprise a system. The result obtained from this intermediate measure of function types is called *function counts*. Function counts are further adjusted by a measure of *environmental complexity*. The mathematical definition of function points is shown below:

$$FUNCTION\ POINTS = FUNCTION\ COUNTS * (.65+(.01*\sum_{F=1}^{14} COMPLEXITY_f))$$

where

FUNCTION COUNTS	= *instances of the five function types;*
COMPLEXITY	= *a complexity factor, f, associated with each of fourteen descriptors of the implementation complexity of a system.*

A major concern in traditional development environments is calibrating the people who carry out the function point analysis. Our experience in a recent study of software development productivity suggested that even when a group of well-trained individuals performs function point analysis for the same set of software projects there are bound to be discrepancies which have to be resolved (BANK90A). Individual differences in interpreting the documentation, knowledge of an application and experience in conducting function point analysis can all drive these differences. In addition, recent research by Low and Jeffrey (LOW90) found that significant training in the use of the complexity measures is necessary to ensure that the correct constructs are being measured.

2.2. Function Points and Integrated CASE

Unfortunately, none of these problems disappears when applications developed using integrated CASE tools are examined using function point analysis. While the quality of the documentation is much improved due to its automation and storage on a central repository, utilizing such documentation is still a very costly and time-consuming process. Although a major source of the power of CASE tools comes from their 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. This is unlikely to closely match what a person would write. Thus, analyzing CASE-generated code would be an onerous and, most likely, an inefficient task that would require a large amount of time and effort to do well.

Other potentially more serious problems that need to be considered deal with the content of the function points method. For example, the classification scheme used in the identification of FUNCTION COUNTS is not intuitive for ICE-developed software. The components of the function points procedure (inputs, outputs, external interfaces, queries and files) do not follow naturally from the building blocks of this CASE development environment. The CASE methodology used in ICE development enforces modularization of application code and object-oriented design, which both promote more efficient system development and maintainability. But 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 within a module or an object, resulting in a subjective classification of function counts and low consistency in the function points estimated by different analysts.

Straightforward identification of function types via procedures standard in 3GL development environments is also prone to double-counting the labor consumed in developing systems with CASE. Since an important feature of ICE is its *central repository*, significant opportunities exist to reuse code which adds to the functionality a system delivers without requiring much additional effort. So, even though counting the five function types remains an exhaustive classification of the size of the product, reuse must be factored in when function points are being used estimate the labor required to build an ICE application.

Classification of FUNCTION COUNTS into simple, average and complex levels of complexity according to the function points methodology is also problematic. The weights applied to the different complexity levels were determined by Albrecht by trial and error (ALBR79). However, Symons (SYMO88) concluded that a new set of weights might need to be recalibrated for any new technology, new organization, or new development atmosphere. Clearly CASE qualifies as a technology which will require Albrecht's weights to be recalibrated.

The rationale for decomposing function types into simple, average and complex was based on the observation that they required a different amount of time to code. However, in ICE development the ratio between the time required to code a simple type and a complex type may not be as large as it wa~ in traditional development environments. Thus the complexity classification used in the function point analysis method may not do as well in estimating the actual level of software development labor consumed.

In the integrated CASE environment we have been studying, reuse affects effort far more than any other factor, and may also contribute to productivity gains in the testing and implementation stages of the system development life cycle (BANK90A). Reused objects will have been tested in other applications previously. Reuse, together with the availability of the automatic code generation facility, may reduce the development labor required to incorporate higher levels of complexity measured by the subjective COMPLEXITY FACTORS of the function points method. CASE utilities for graphics generation and screen painting are good examples that can produce major time savings for developers. As a result, whether the COMPLEXITY FACTORS remain the relevant dimensions by which to adjust FUNCTION COUNTS is an open question.

2.3. Measuring Code Reuse

Since most studies of code reuse in traditional development environments have concentrated on the problems of encouraging it, rather than identifying and measuring it, it is not surprising that there are few rigorous definitions of reuse in a systems development performance evaluation context (LANE84, NUNA89, RAJ89). *Reuse*, as the name implies, is the employment of previously written code as an alternative to writing new, possibly identical, code to perform the same or a similar function.

The level of code reuse may be computed as the number of times a particular piece of code, data element or object is reused within the context of a program, application or information system (POLS84). As Hall (HALL87) has pointed out, however, this intuitive measure does not, in itself, address many of the managerial questions concerning code reuse. Some of the key questions include:

* What portions of the code are being reused, and among those which pieces are reused most often?

* What is the impact of such reuse on productivity and development costs?

* How effective is a particular system or environment in promoting code reuse to reduce development costs?

Standish (STAN84) and Neighbors (NEIG84) provide useful perspectives on the measurement of code reuse for 3GL development environments. Standish proposed that reuse be measured at the line of code level. This approach suffers from the disadvantages endemic to source line of code metrics: they are conceptually simple, but are unlikely to convey managerially useful information, and they say nothing about the functionality of a system. Neighbors argued that reuse should be abstracted from the level of source code into some meta-language which relates more closely to the problem. This approach is likely to be of practical use in CASE environments such as ICE, where there is a high-level representation of the system.

Gaffney and Durek (GAFF89) modeled the impact of code reuse as a function of the relative costs of new and recycled code, and of their relative incidence. The authors' analysis suggests a strong rationale for creating code reuse metrics which support economic modeling of software development productivity and measurement of the business value of CASE technology. We next turn to a discussion of some new metrics which address the problems discussed in this section.

3. PROPOSAL: RE-THINKING SOFTWARE METRICS FOR OBJECT-ORIENTED INTEGRATED CASE DEVELOPMENT ENVIRONMENTS

An object-oriented integrated CASE environment presents an interesting opportunity to examine new metrics for measuring software development performance. ICE offers support for the creation and automation of a new set of software metrics through the following features:

* the use of objects as application building blocks provide a natural means by which to measure reuse;
* a central repository which stores all objects enables a historical record of the development of an application to be maintained;
* the storage of an abstract object hierarchy in the repository defines the functionality of an ICE application, and this object hierarchy can be mapped into the function point analysis methodology.

3.1 Object-Oriented Development in Integrated CASE

The central repository in ICE stores information about different kinds of objects used in applications developed with the tool. Examples of object types defined for the CASE tool include: RULE SETS, 3GL MODULES, SCREEN DEFINITIONS and USER REPORTS. Each object type is defined precisely and rigorously in order to make the process of software development conducive to object reuse. A RULE SET contains most of the instructions that observers unfamiliar with CASE would call "the program". A 3GL MODULE is a pre-compiled procedure, originally written using 3GL. A SCREEN DEFINITION is the logical representation of an on-screen image. A USER REPORT means the same thing as it does in development environments other than ICE.

All objects associated with an application are functionally organized into an *object hierarchy*. An application consists exclusively of these objects and each application can be identified by a high-level BUSINESS PROCESS, which calls other RULE SETS. These RULE SETS 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 illustrates these concepts.

INSERT FIGURE 1 ABOUT HERE

The relationships between objects (which RULE uses which 3GL MODULE, which invokes which SCREEN, etc.) are themselves stored in the central repository. Collectively, the set of object instances and relationships between them make up the meta-model of the application, and this can be used to identify the objects comprising an application. Identification of such objects provides two important benefits. *First*, it follows the natural building process of CASE systems and is therefore intuitive and has the potential to be more accurate and consistent. *Second*, the meta-model in the repository can be utilized to automate the identification of objects. This would lead to considerable savings in the effort and cost involved in collecting information about the objects used, and motivate implementation of the revised measurement procedures we will shortly describe.

3.2. Measuring Code Reuse for Integrated CASE

We propose that reuse in CASE development environments which are similar to ICE be measured in three separate classes of metrics: *reuse leverage*, *reuse value*, and *reuse classification*. Each of the proposed metrics relies on being able to identify objects that are reused, rather than lines of codes or modules of an application. Each also provides answers to different kinds of questions managers may have in controlling the performance of software development projects.

Reuse leverage metrics measure the number of times that objects are used within a system. We define the degree of reuse within a system as:

$$\textit{REUSE LEVERAGE} = \frac{\textit{TOTAL NUMBER OF OBJECTS USED}}{\textit{NUMBER OF NEW OBJECTS BUILT}}$$

This measure of reuse can be used at several levels of analysis, for example, in aggregate form for all the objects in an application, or by object type such as RULE SETS, SCREEN DEFINITIONS or USER REPORTS. The primary value of such metrics is that they enable an analyst to identify what is being reused and how much reuse is occurring.

To measure the actual productivity gains associated with code reuse, we must distinguish between the reuse of easily-programmed objects and the reuse of more costly objects. We can compute *reuse value* by weighting the level of reuse by the cost of programming the various types of objects that are reused. So, rather than just counting objects, we add up the *cost* of each object to form the following ratio:

$$\textit{REUSE VALUE} = 1 - \frac{\sum_{j=1}^{J_k} COST_j}{\sum_{j=1}^{J} COST_j}$$

where

$COST_j$	= *the standard cost in person days of building object j;*
J	= *the total number of occurrences of objects in an application meta-model hierarchy;*
k	= *the total number of unique objects built for this application.*

We normally would include in our computation of code reuse any object which is found in the repository, rather than rewritten from scratch. But for some managerial purposes, it may be useful to classify reuse as internal and external. *Internal reuse* refers to code reuse within a system or subsystem, as defined by its meta-model hierarchy. *External reuse* refers to the reuse of objects which are in the repository, but which currently belong to a different system, and were originally developed for it. While both kinds of reuse are valuable, different managerial policies may be required to encourage them.

For example, the degree of internal reuse will probably depend upon the size of the team developing a given application, and the quality of the communications within that team. The degree of external reuse, on the other hand, will depend more upon the quality of the indexing system used to help programmers to identify existing objects which they might be able to reuse.

3.3. Object Points Analysis for Integrated CASE

To explore the questions raised in Section 2 about the usefulness of function points for estimating software development labor for ICE projects, we conducted two sets of interviews with managers and analysts experienced in the use of the tool. The first set of interviews was in the form of Delphi sessions. Small groups of project managers were asked to individually estimate the time required to build a small application involving several different levels of functionality, and then attempt to achieve a group consensus. The Delphi sessions were taped, and later analyzed for themes that unified the discussions and led to the group estimates.

Based on the analysis, we conducted a second set of individual follow-up interviews with project managers responsible for developing and estimating projects. The interviews included more focused questions regarding how they might estimate using objects as the basis of their estimation. This enabled us to identify the usefulness of an output measurement and estimation approach that is more closely linked to the ICE development environment. Our analysis indicated that project managers employ estimation heuristics which rely on the number of different types of objects that need to be developed for a project. For example, using these heuristics, a project manager initially estimates the number of RULE SETS, 3GL MODULES, SCREEN DEFINITIONS and USER REPORTS that will comprise the final application software.

The project managers' responses in our interviews raised a number of important questions regarding the relationship between function points and this new proposal. For example,

* Do objects really capture the user functionality of an ICE application?
* Does knowing the number and types of objects comprising a system provide sufficient information to estimate the labor required to build it?
* Is knowing the number and types of objects only useful as a first approximation for development labor, or is it useful as a means to gauge productivity after application development has been completed?

Continuing the interviews, we also learned that similar to the function types in function points, different objects exhibit different levels of complexity and functionality. Thus, they also require different amounts of development labor to construct. A synthesis of our project manager interview results enabled us to classify 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 heuristics are summarized in Table 1 below in terms of the *average time required to build a given object type*.

INSERT TABLE 1 ABOUT HERE

The *means* of the project manager responses are shown in the table.

Two new output measures are suggested by our analysis. The first, termed *object counts*, is determined by summing the occurrences of individual objects of the four types. The second, called *object points*, is defined as follows:

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

where

OBJECT-EFFORT-WEIGHT	= *average estimated development effort associated with object type t, based on project manager heuristics;*
OBJECT-OCCURRENCE	= *number of occurrences of one of four object types t (including RULE SET, 3GL MODULE, SCREEN DEFINITION and USER REPORT) in an ICE application.*

Object points can be further refined to incorporate information that distinguishes among the three levels of complexity for each object type in terms of the labor required. At this time, we have not yet developed a mechanism to classify objects according to these complexity levels that does not involve significant manual effort on the part of the analyst. A deeper investigation into the nature and use of heuristics for estimation and classification of objects in ICE environments is required in order to specify the dimensions of object complexity.

Use of heuristics by experts for the estimation of software development costs has been reported previously in other development environments (VICI89). However, note that the identification of objects in the context of ICE presents no problems because of the availability of the central repository and the application meta-model. In fact, identification of these objects and classification into different complexity should be readily automatable.

Thus, we conclude that major opportunities exist to pioneer new metrics that are more closely related to the software development environments they are meant to describe, yet general enough to be captured for different kinds of CASE tools.

4. JUSTIFICATION: MANAGING THE CASE DEVELOPMENT PROCESS USING AUTOMATED METRICS

4.1. The Automation of Function Point Analysis

We can use the application meta-model that is stored by ICE to identify the objects associated with any application system. Since the meta-model is hierarchical, following the chain of relationships will reliably lead us to all the objects which may be accessed or invoked by a given object. Traversal of the hierarchy of RULE SETS which comprise an application, or sets of applications, is a very powerful capability that can be exploited in the design and development of automated software metrics facilities for ICE.

Clearly, any attempt to automate the collection of software metrics in ICE begins with a major advantage over similar efforts in third-generation environments. Much of information which is needed to calculate a variety of software metrics (code reuse, function points, object points, etc.) is already contained in usable form in the application meta-model. In traditional environments, this task must be accomplished on the basis of documentation, which is rarely complete or up-to-date, and software naming conventions which, even when they are followed, rarely identify the use of code by multiple applications.

The design of ICE's object-oriented development language is such that a precise mapping may be defined between each object and its associated functionality. In traditional environments, the only way to perform the mapping between programs and functionality is to manually figure out what each program is doing, again with the aid of such documentation as may exist.

Of the five function types used in the computation of function points, four measure data flows that either enter or leave the "boundary" of an application. Internal files constitute the fifth function type; they measure data stores internal to the application. Object and entity relationship definitions may be decomposed into specific functional roles. This "mapping" enables function counts to be identified. This is illustrated in Figure 2, which also provides a conceptual representation of what we mean by the "application boundary." (For additional details on the content of the mapping between functions and ICE objects, see BANK90B.)

INSERT FIGURE 2 ABOUT HERE

Using this approach we developed an *automated function point analyzer* with three main components: an Object Identifier, a Function Counter and a Complexity Factor Counter. The general architecture of the automated function point analyzer is shown in Figure 3.

INSERT FIGURE 3 ABOUT HERE

The *Object Identifier* traverses the meta-model in order to identify all the objects used in an application that have to be evaluated for functionality. It starts with a BUSINESS PROCESS or high-level RULE SET chosen by the project manager that defines the application or part of the application being analyzed, and navigates the hierarchy downward until all relevant objects have been found.

The *Function Counter* performs the mapping described in the previous section from objects and entity relationships, to function types and complexities, and then to function counts.

The *Complexity Factor Counter* computes environmental complexity, which is used in function point analysis as an adjustment factor, to allow for the overall complexity of the task being implemented and the environment within which it is being implemented. A point score is assigned to each of fourteen complexity factors, and the total of these scores is the complexity factor.

The function point analyzer determines the scores for fourteen complexity factors through a combination of objective, automated measures and online inputs provided by project managers familiar with the technical aspects of implementation. In the current implementation, the objective measures are computed in parallel with managers' inputs, which only take a few minutes. When they have been sufficiently validated, the corresponding manual inputs will be replaced entirely, where possible. Each complexity factor also has a separate input response screen that displays a definition of the complexity factor. This can help a project manager who may not be familiar with function point analysis to give accurate and consistent responses.

4.2. Extensions: Automating Reuse and Object Point Collection

An integrated, object-oriented CASE environment similar to ICE also provides a major assist for the implementation and control of code reuse. *First*, ICE code exists at a level of granularity more conducive to the implementation of code reuse. While it is rare that an entire 3GL program will prove to be reusable, such programs frequently contain routines which *could* be reused, with a little modification, were the programmer aware of their existence. An object-oriented system may be designed so that each such routine is a unique object. This makes reuse opportunities considerably easier to identify and to exploit.

Second, the integrated environment serves to support the control and the measurement of code reuse. With the design of the entire system stored centrally along with the software itself, an instance of code reuse becomes readily identifiable as the repeated invocation of an object within the repository.

To follow up on these ideas, we designed an automated code reuse analyzer for use within ICE. The tool analyzes an existing software application, reporting the levels of reuse for the various elements comprising the application. The code reuse analyzer shares many features in common with the function point analyzer. For instance, it identifies all the relevant objects for a given analysis by systematically navigating the hierarchy of calling relationships within the repository. (In fact, it even reuses much of the code required to develop the function point analyzer.) Once all the objects within an application have been identified, and the instances of reuse have been noted, a range of managerially useful code reuse metrics can be computed.

An object point analyzer would also operate along the same lines. Automating the analysis of an ICE application for function points and code reuse delivers a portion of the object point information for free. This is accomplished when the Object Function Table (shown in Figure 3) is instantiated based on the function point analyzer's scan of the object repository application meta-model. This in turn results in information about the typed object count for an application. The missing piece is that function point analysis does not require estimation of the effort that is required to build objects of different levels of complexity. As a result, this capability must be added to deliver a fully automated object point analysis. For the current implementation, we are investigating the usefulness of the project manager averages reported in Table 1, because they can be readily used to weight the objects stored in the Object Function Table to obtain a short form estimate for object points.

4.3. Software Development Life Cycle Management Via Automated Metrics

The primary benefits of automated software metrics come from the opportunities created for management to gain new insights into the performance of a firm's software development organization. Previously, obtaining a *point estimate* (i.e., at one point in time) of development productivity for a project required significant effort. The correct documentation had to be obtained, development labor information had to be pieced together, and then finally the analysis had to be performed. But, point estimates only describe the outcome of development activities; they fail to describe the process leading to the delivery of completed software. Yet it is the process that management needs to fine-tune so that the outcome can be improved.

With automated software metrics, it is possible for management to replace point estimates of productivity with a full software development life cycle *trajectory estimate* of performance. For example, upon reaching significant project development milestones, automated measurement of function points, object points and various reuse metrics can be carried out, based on the software stored by the repository at that time. Additional measures can also be made on demand when management has specific questions about the development performance of a specific project.

Software development life cycle performance trajectory estimates can be made following the natural course of the development of a software application. For example, at the inception of a project, very little will be known about what the software finally will look like, but there will be significant information available about the kinds of components that are required to achieve such functionality. Order of magnitude estimates will be required to identify the overall costs associated with going ahead with a project. A rough estimate of the number of objects can be made prior to the start of the creation of the functional design of a system. This estimate can be refined further as the project progresses through technical design. By this time, however, it will be possible to obtain information from the repository about the future contents of the application, though they may not be entirely built.

Automated software metrics will be even more useful as a project moves into the construction and testing phases. Figure 4 below depicts the quarterly progress of two projects (marked A and B) in terms of metrics that can be captured automatically: function points per person month and the observed level of reuse leverage.

INSERT FIGURE 4 ABOUT HERE

If a simple average productivity rating were assigned to the two projects, Project A would clearly exhibit a higher level of productivity overall (in terms of function points per person month) than Project B. Yet without the additional information provided by the trajectory shown in the upper graph of Figure 4, important information would be lost to management. For example, note that

A's productivity is maximized in the middle of the construction phase, when the project was likely to have been fully staffed, but it fell later, perhaps due to implementation problems, the slippage of deadlines, changes in the development environment, or interference from new projects that were taking more of management's time. In addition, B's productivity met or exceeded the targeted minimum in only four of the eight quarters.

Coupling this information with the reuse leverage trajectory shown in the lower graph of Figure 4 provides additional information. Note that the targeted level of code reuse leverage was only met in one of the eight quarters. Although these graphs do not provide a complete picture of what was occurring as the applications were being developed (for example, it is possible that the more productive project was much larger and provided more opportunities to make effective use of the CASE tool), they nevertheless suggest the possibility that management needs to take additional steps to manage code reuse to avoid substandard development productivity results.

Although the example we have used is highly simplified, other useful comparisons of descriptive project performance metrics would be possible to pave the way for a fuller understanding of the dynamics of software development. For example:

* reuse classification metrics can be used to indicate the extent of the extra gains derived from external reuse;
* both function points and object points can be tracked to identify the performance of each in estimating the final level of development labor required;
* projects can be compared over time for baseline changes in the level of productivity observed, as the firm's use of a CASE tool matures and new capabilities become available, and as the shape of the firm's project performance trajectories changes;
* the evaluation of project managers can also be tied to trajectory measures of project performance;
* productivity and reuse metrics for the full development life cycle can later be used to identify the effects of other variables including team size, experience levels,

developer training and the size of an application.

We believe that tracking the software development life cycle of an ICE application with automated performance trajectory metrics offers management the chance to obtain a comprehensive understanding of a firm's software development operations. As our sketch of the function point analyzer suggested (see Figure 3), a variety of metrics can be obtained for use by individuals with different levels of management responsibility, for example, a project manager, the vice president of systems development or the chief information officer. Having such information available to these management levels can lead to better decisions to control the strategic costs of large scale software development, and offer the firm a new range of opportunities to achieve competitive advantage.

5. CONCLUSION: RESEARCH AGENDA

Senior managers of software development operations broadly agree on the need for metrics and measurement programs that:

* provide an accurate picture of development productivity across different kind of applications and development environments;
* enable standards to be devised to identify efficient and inefficient projects;
* support performance tracking without incurring high costs or affecting development activities directly;
* help to support the argument that operating an effective, high technology software development operation can assist management in its efforts to maximize the value of the firm.

Having informative and readily implemented metrics available for use in integrated CASE environments can help to jump start software development performance measurement programs in firms that have been reluctant to measure. More importantly though, with the ability to automate much of the effort of collecting software metrics, tracking performance across the entire development life cycle of a project becomes possible. This also increases the likelihood that additional investments in CASE will be made by management for the right reasons.

Our exploratory work on the use and automation of the metrics discussed in this paper is still in progress, so a number of important questions remain for future research. For example, with respect to function point and object point analysis, additional work is required to investigate the estimation performance of object points in development environments other than the firm that has been the field site to date. And, once we are able to obtain a steadier stream of data on developed and developing projects from the automated function point analyzer, it will be possible to study whether object points perform better as early life cycle estimators or as a more robust output metric than function points for the integrated CASE environment.

A second major set of issues that need to be investigated in more detail is how reuse should be managed to maximize productivity, in view of the information provided by the new reuse metrics. In our preliminary investigation of code reuse, we have found that developer experience plays an important role in the delivery of projects which exhibit high levels of reuse. A major question, then, is how repository objects should be indexed to encourage easy identification and reuse by developers. Another question which future research should address is the extent to which targeted levels of code reuse can be mandated, and the effects this would have on development productivity.

Many observers have speculated that the real potential of CASE is to increase software quality and functionality. The ICE applications we studied deliver a very high level of functionality compared to their 3GL counterparts, and in some cases, direct comparisons are inappropriate because the quality of the product is so different. Thus, the final item on our research agenda is to determine the extent to which the creation of software developed using integrated CASE actually delivers software with a higher overall value to the firm. Clearly, the main issue here is not user satisfaction. Instead, the key question for management will be: What is the long-term business value of the increases in functionality delivered by CASE-developed software? Thus, a major challenge that remains is to devise an evaluative approach that yields metrics which assist in translating software functionality into the various dimensions of value obtained by the firm.

REFERENCES

ALBR79	Albrecht, A. J., Measuring Application Development Productivity. In *Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium*, IBM (October 1979), pp. 83-92.
ALBR83	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), pp. 639-647.
BANK90A	Banker, R. D., and Kauffman, R. J. An Empirical Assessment of Computer Aided Software Engineering (CASE) Technology, A Study of Productivity, Reuse and Functionality. Working Paper, Center for Research on Information Systems, Stern School of Business, New York University (March 1990).
BANK90B	Banker, R. D., Kauffman, R. J., and Kumar, R. Output Measurement Metrics in a Computer Aided Software Engineering (CASE) Environment: Critique, Evaluation and Proposal. *Management Information Systems Quarterly* (forthcoming).
BANK90C	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 (June 1990).
BANK90D	Banker, R. D., Kauffman, R. J. and Zweig, D. Metrics for the Code Reuse in Software Development. In preparation (1990).
BOUL89	Bouldin, B. M. CASE: Measuring Productivity -- What Are You Measuring? Why Are You Measuring It? *Software Magazine,* 9:10 (August 1989), pp. 30-39.
DREG89	Dreger, J. B. *Function Point Analysis*. Prentice-Hall, Englewood Cliffs, NJ (1989).
FISH90	Fisher, J. T. IBM's Repository: Can Big Blue Establish OS/2 EE As the Professional Programmer's Front End? *DBMS* (January 1990), pp. 42-49.

GAFF89 Gaffney, J. E., Jr., and Durek, T. A. Software Reuse -- Key to Enhanced Productivity: Some Quantitative Models. *Information and Software Technology* 31, 5 (June 1989), pp. 258-267.

GOLD89 Goldberg, A., and Pope S. T. Object Oriented Programming is Not Enough. *American Programmer: Special Issue on Object-Oriented Observations* 2, 7-8 (Summer 1989).

HALL87 Hall, P. A. V. Software Components and Reuse -- Getting More Out of Your Code. *Information and Software Technology* 29, 1 (January-February 1987), pp. 38-43.

HAZZ89 Hazzah, A. Making Ends Meet: Repository Manager. *Software Magazine* (December 1989), pp. 59-72.

JONE84 Jones, T. C. Reusability in Programming: A Survey of the State of the Art. *IEEE Transactions on Software Engineering* SE-10, 5 (September 1984), pp. 484-494.

KEME89 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*, Kona Lua, Hawaii, IEEE (January 1989).

LANE84 Lanergan, R. G. and Grasso, C. A. Software Engineering with Reusable Designs and Code. *IEEE Transactions on Software Engineering* SE-10, 5 (September 1984), pp. 498-501.

LOW90 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 1, 1990), pp. 64-71.

MENG90 Meng, B. Object Oriented Programming. *MacWorld* (January 1990), pp. 174-180.

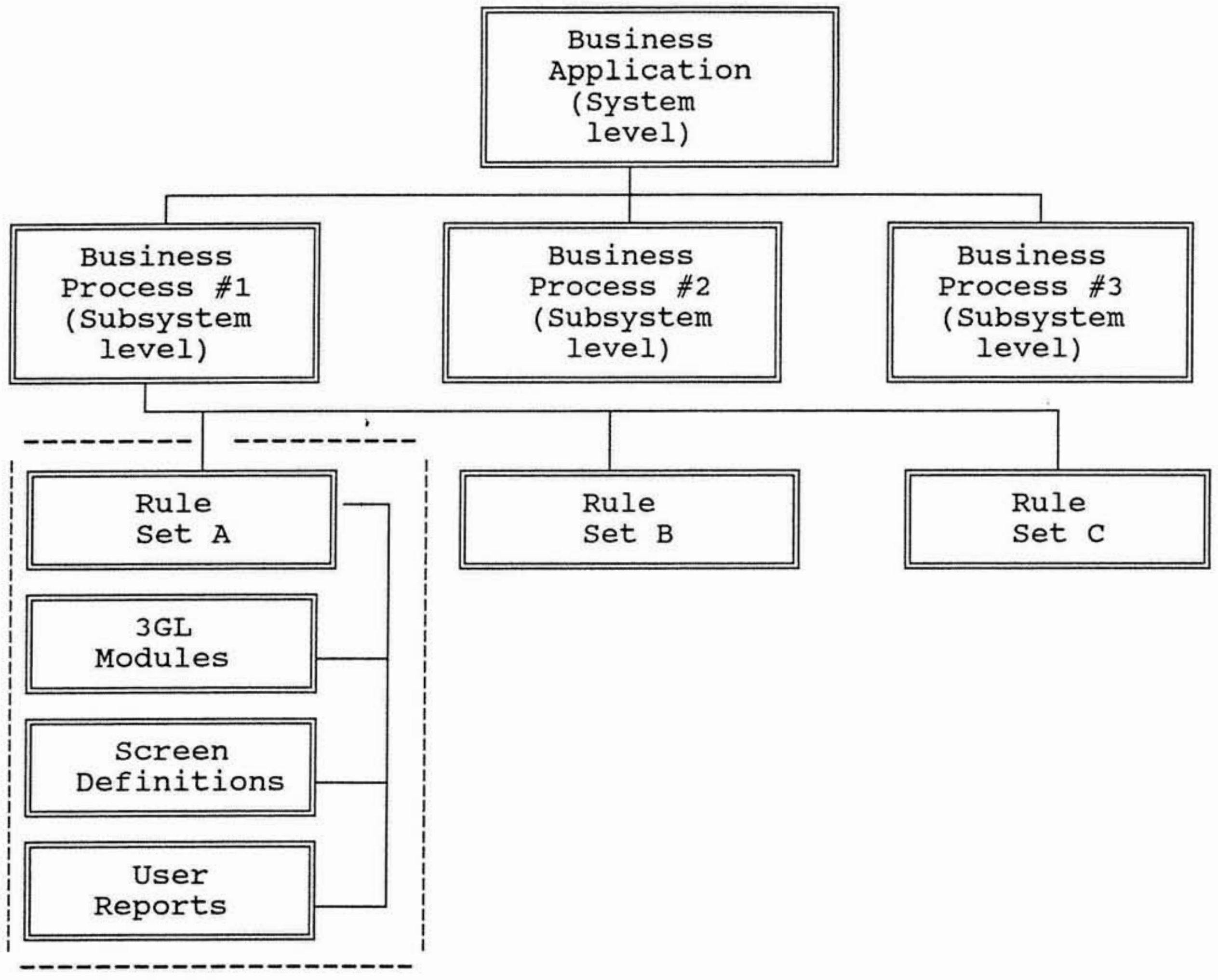
MEYE87 Meyer, B. Reuse: The Case for Object-Oriented Design. *IEEE Software* (March 1987), pp. 50-64.

MEYE88 Meyer, B. *Object Oriented Software Construction*. Prentice Hall, New York (1988).

NEIG84 Neighbors, J. M. The DRACO Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering* SE-10, 5 (September 1984), pp. 564-574.

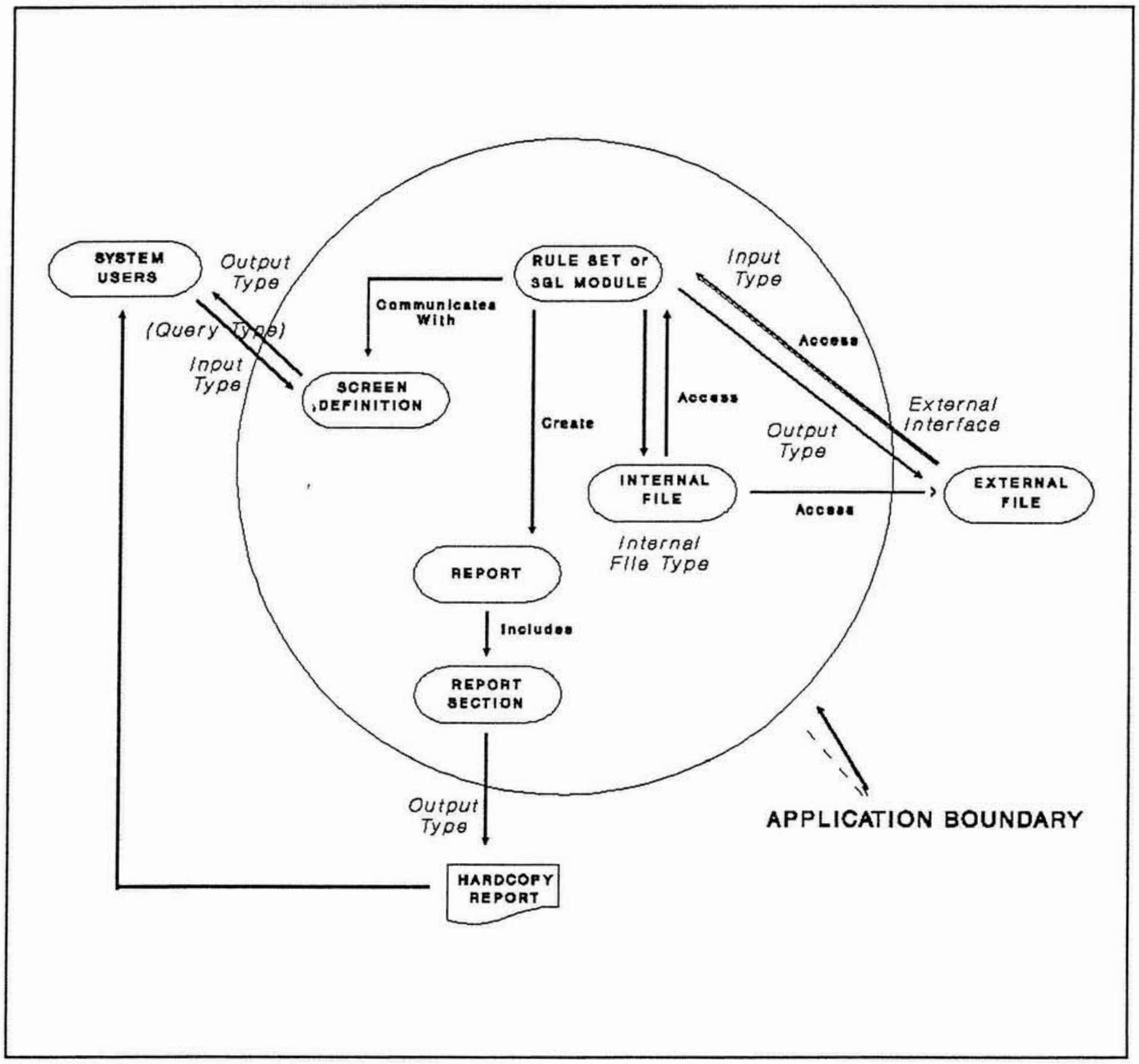
NUNA89	Nunamaker, J. F. Jr., and Chen, M. Software Productivity: A Framework of Study and an Approach to Reusable Components. In *Proceedings of the 22nd Hawaii International Conference System Sciences*, IEEE, Kona Lua, Hawaii (January 1989, pp. 959-968.
POLL90	Pollack, A. The Move to Modular Software. *New York Times* (Monday, April 23, 1990), pp. D1-2.
POLS84	Polster, F. J. Reuse of Software Through Generation of Partial Systems. *IEEE Transactions on Software Engineering* SE-10, 5 (September 1984), pp. 402-416.
RAJ89	Raj, R. K. and Levy, H. M. A Compositional Model for Software Reuse. *The Computer Journal* 32, 4 (April 1989), pp. 312-323.
SENN90	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).
STAN84	Standish, T. A. An Essay on Software Reuse. *IEEE Transactions on Software Engineering* SE-10, 5 (September 1984), pp. 494-497.
SYMO88	Symons, C. R. Function Point Analysis: Difficulties and Improvements. *IEEE Transactions on Software Engineering* 14, 1 (January 1988), pp. 2-10.
VICI89	Vicinanza, S., Mukhopadhayay, 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 (1989)
ZWAN84	Zwanzig, K. *Handbook for Estimating Function Points*. GUIDE Project -- DP-1234, GUIDE International (Nov. 1984).

Figure 1. A Simplified Repository-Based Application Meta-Model


Business Application (System level)
Business Process #1 (Subsystem level)
Business Process #2 (Subsystem level)
Business Process #3 (Subsystem level)
Rule Set A
Rule Set B
Rule Set C
3GL Modules
Screen Definitions
User Reports

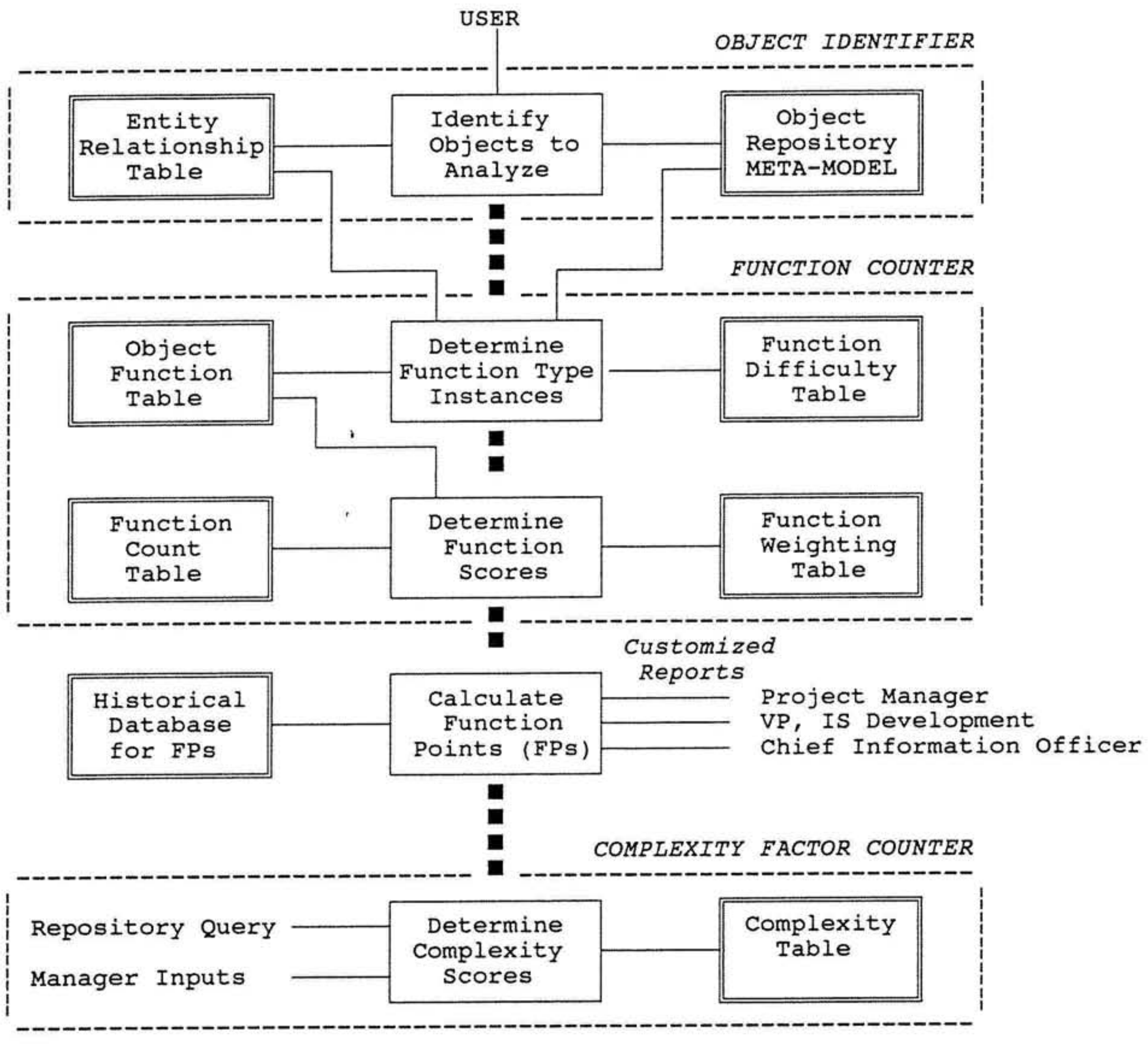
An application is represented in ICE by a menu of BUSINESS PROCESSES and all the objects called by these BUSINESS PROCESSES. The first step in analyzing a system is to identify these objects, by iteratively tracing the calling relationships stored in the application meta-model. A BUSINESS PROCESS will call one or more RULE SETS. Each RULE SET, in turn, may call other RULE SETS, 3GL MODULES or other ICE objects. Note that the use of an object by an application system does not preclude its reuse by another application. Nor does the occurrence of an object within an application's object hierarchy imply that the object was originally created for use in that application.

Figure 2. Mapping from ICE Objects to Function Counts



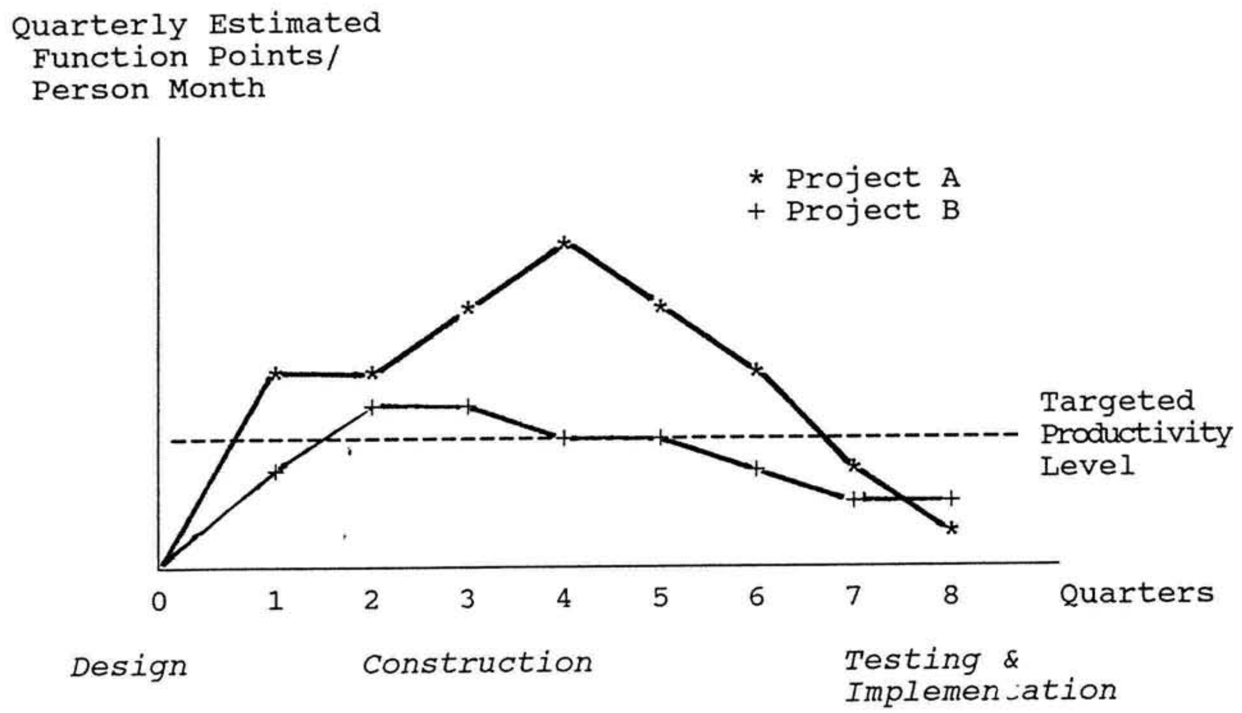
Function point analysis measures the functionality a system delivers to the user in terms of data transfers into or out of that system (Inputs, Outputs, Queries, External Interfaces), and in terms of the data stores (files) used. A 3GL program normally contains all five function classes. An ICE object, however, is severely constrained in the functionality it can represent, to the points where a system's function count can be computed by identifying and classifying its objects.

Figure 3. The Automated Function Point Analyzer: A Schematic


USER
OBJECT IDENTIFIER
Entity Relationship Table
Identify Objects to Analyze
Object Repository META-MODEL
FUNCTION COUNTER
Object Function Table
Determine Function Type Instances
Function Difficulty Table
Function Count Table
Determine Function Scores
Function Weighting Table
Customized Reports
Historical Database for FPs
Calculate Function Points (FPs)
Project Manager
VP, IS Development
Chief Information Officer
COMPLEXITY FACTOR COUNTER
Repository Query
Manager Inputs
Determine Complexity Scores
Complexity Table

The *function point analyzer* consists of three subsystems. One uses the meta-model to identify the objects in the application under analysis. The second uses it to assign function count scores to those objects. The third obtains task complexity measures, and may require programmer or manager input in parallel with the automated analysis.

Figure 4. Software Development Performance Trajectories: Function Points and Reuse



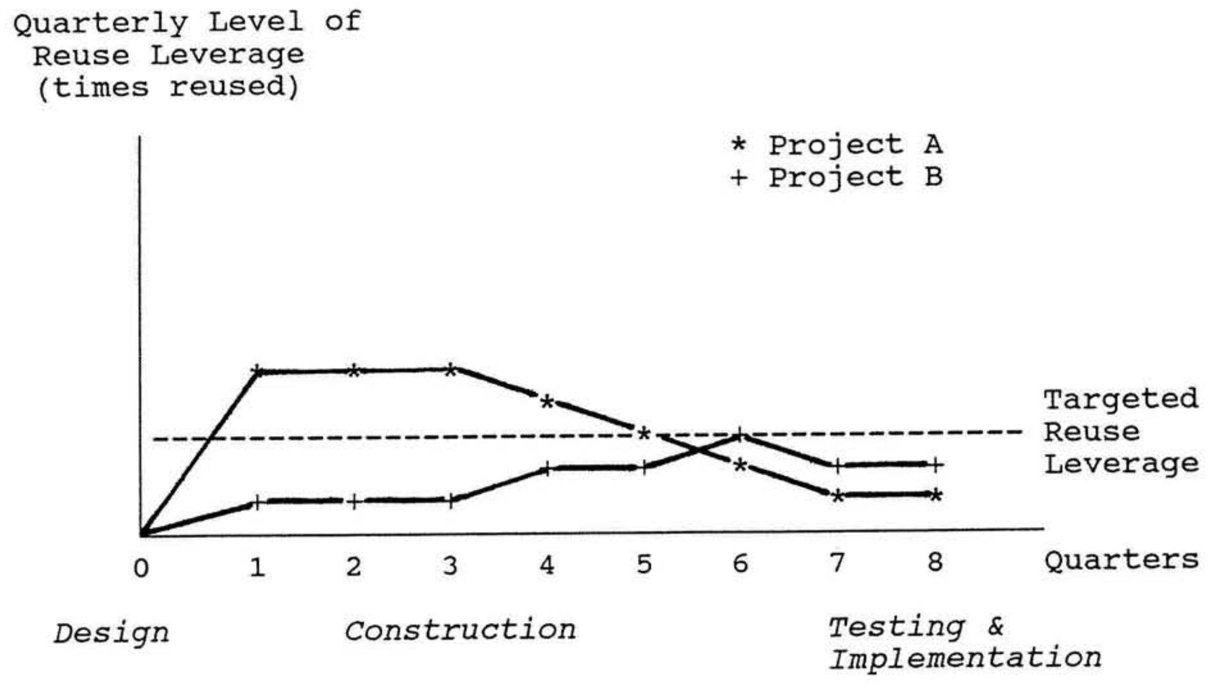


Table 1. Project Manager Development Effort Heuristics

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