AUTOMATED SOFTWARE METRICS, REPOSITORY EVALUATION AND SOFTWARE ASSET MANAGEMENT: NEW TOOLS AND PERSPECTIVES FOR MANAGING INTEGRATED COMPUTER AIDED SOFTWARE ENGINEERING (I-CASE)

by

Rajiv D. Banker

Robert J. Kauffman

AUTOMATED SOFTWARE METRICS, REPOSITORY EVALUATION AND SOFTWARE ASSET MANAGEMENT: NEW TOOLS AND PERSPECTIVES FOR MANAGING INTEGRATED COMPUTER AIDED SOFTWARE ENGINEERING (I-CASE)

by

Rajiv D. Banker
Arthur Andersen Chair in Accounting and Information Systems
Carlson School of Business
University of Minnesota

and

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

May 1991

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

Working Paper Series

STERN IS-91-8

An earlier version of this work was presented at the *Workshop on IT Strategy and Architecture*, Wharton School of Business, University of Pennsylvania, Philadelphia, PA, June 22, 1990.

The authors 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 thank Gig Graham and Michael Oara for their recent assistance. Our colleagues Eric Fisher, Charles Wright and Dani Zweig contributed 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 #SES-8709044, and Rachna Kumar for useful comments on prior drafts of this paper. All errors in this paper are the responsibility of the authors.

KEYWORDS: CASE, code reuse, computer aided software engineering, function point analysis, integrated CASE, I-CASE, object-based, repositories, software asset management, software costs, software development, software metrics, productivity.

AUTOMATED SOFTWARE METRICS, REPOSITORY EVALUATION AND SOFTWARE ASSET MANAGEMENT: NEW TOOLS AND PERSPECTIVES FOR MANAGING INTEGRATED COMPUTER AIDED SOFTWARE ENGINEERING (I-CASE)

ABSTRACT

Automated collection of software metrics in *computer aided software engineering* (CASE) environments opens up new avenues for improving the management of software development operations, as well as shifting the focus of management's control efforts from "software project" to "software assets" stored in a centralized repository. Repository evaluation, a new direction for software metrics research in the 1990s, promises a fresh view of software development performance for a range of responsibility levels. We discuss the automation of function point and code reuse analysis in the context of an *integrated CASE* (I-CASE) environment deployed at a large investment bank in New York City. The development of an automated code reuse analysis tool prompted us to re-think how to measure and interpret code reuse in the I-CASE environment. The metrics we propose describe three dimensions of code reuse -- leverage, value and classification -- and we examine the value of applying them on a project and a repository-wide basis.

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 costs of software development. But such investments also raise many questions for management (BOUL89, SENN90). For example, what are the features of an integrated CASE (**I-CASE**) tool that enable a firm to maximize development productivity across the entire systems development life cycle, while maintaining acceptable quality and functionality? How does a CASE tool affect the activities associated with different phases of the software development life cycle? Are the benefits balanced, or are they concentrated in analysis and design rather than construction and testing? Does the storage of reusable code in a firm's centralized repository create an appropriate amount of leverage to power up software development productivity? Are the benefits of CASE sufficient to justify the high costs of implementing it? Is the move to modular, *object-based* software development paying off (POLL90)?[1]

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 emerging I-CASE environments of the 1990s. In this paper, we present a vision for "software asset management" that differs from the traditional focus on software development project management (KARI90). Instead, its focus is on the contents of the repository of software -- the firm's unique software assets -- and how the repository can be evaluated using automated facilities to measure a range of new metrics that will help management to gain insight into the leverage that CASE can create. This "software asset management" perspective is based on our experience in evaluating an I-CASE environment that has provided major gains in software development productivity (BANK90A).

2. FINDINGS FROM OUR RECENT RESEARCH ON INTEGRATED CASE

Our research on development productivity gains associated with "High Productivity Systems" (HPS), a large-scale I-CASE tool deployed at the First Boston Corporation and Seer Technologies led to a number of interesting discoveries. (For a brief overview of the content of the tool and the rationale behind its implementation, see Sidebar 1.)

[1]"Object-based" refers to the use of objects in modular application design, without the requirements that are associated with the "object-oriented" paradigm. In addition to object-based modular application design these include: the use of abstract data types, the deallocation of memory space for unused objects without programmer intervention, the use of "object classes," and an "inheritance mechanism" which allows one class of objects to become an extension or restriction of another. (See MEYE88 for an authoritative treatment of the complexities of this approach to software construction.) In addition, Booch (BOOC89) provides additional descriptive information on the evolution of "object-based" design into "object-oriented" design.

INSERT SIDEBAR 1 ABOUT HERE

First, we found that the amount of *code reuse* that occurs in an application represents a significant portion of its functionality, and that traditional metrics fail to take this into account. Code reuse involves using previously written code as an alternative to writing new, possibly identical, code that serves a similar function. Our research has shown that effective code reuse has the potential to be the single most important cost driver during the construction phase of the systems development life cycle. This discovery about the effects of code reuse also led us to consider new models for development productivity in which traditional development productivity measures are adjusted to consider the level of code reuse (BANK90A).

Second, we also learned that there was very little research that has addressed *how* code reuse should be measured. The primary result has simply been not to measure it at all (BANK90D). We believe this to be less an oversight of prior research than a reflection of the realities of 3GL software development. There were few tools to help a developer identify opportunities for reuse, since code was not stored in a central repository. In this environment code reuse could be expected to deliver little leverage.

Third, we learned that the functional organization of an HPS application into objects makes it practical to *automate 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 in the form of an object hierarchy. As we have shown in other research (BANK90C), this hierarchy can be "traversed" or navigated from top to bottom, and this supports the exhaustive identification of all objects comprising an application.

Fourth, we found that the new development environment encourages the use of new approaches to cost estimation associated with application development. 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 HPS applications (BANK90B). In this context, we also found that obtaining metrics that are traditionally used to gauge software development productivity in 3GL environments remains a very costly process when we translated them for use in the HPS development environment, despite the much improved quality of the documentation stored in the repository.

Fifth, and perhaps most interesting, was our discovery that repository-based I-CASE for the first time offers senior managers an opportunity to think about *managing software assets, rather than just software development projects*. (The first example of research that reports on the results of a repository evaluation is contained in Banker, Kauffman and Zweig (BANK90E).) With this "software asset management" perspective in mind, we set out to develop new measurement methods centered around the idea of "repository evaluation." Repository evaluation enables the examination of a firm's changing software assets directly. It can be conducted at the level of a software project, for groups of projects, or for many other levels of analysis that do not involve projects at all. For example, such analysis can be conducted by object type, for objects developed by specific programmers or for repository-wide reuse at a specific point in time.

3. FUNCTION POINT AND CODE REUSE METRICS IN INTEGRATED CASE ENVIRONMENTS

In this section, we examine two kinds of software metrics -- function points and code reuse metrics -- and their role in estimating software size and assessing performance in I-CASE environments.

Function Point Analysis in Traditional Development Environments

The magnitude of a software development effort depends upon several factors: 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 (irrespective of the development language used), and so that software development managers would have a tool to estimate the resources required to build systems of various sizes (ALBR83). An equally important use of function points is as an output metric in the context of software development productivity evaluation. Productivity is normally measured in function points per person month of development labor. (For a brief description of the function point analysis methodology, refer to Sidebar 2.)

INSERT SIDEBAR 2 ABOUT HERE

One 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. 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] Finally, calculating function points is very time-consuming and requires highly-skilled analysts who can understand and interpret imperfect documentation.

Function Point Analysis in I-CASE Environments

Unfortunately, none of these problems disappears when applications developed using I-CASE tools are *measured manually* using function point analysis. Although the quality of the documentation is much improved due to its automation and storage on a central repository, manual function point analysis remains a very costly and time-consuming process. Our research program suggests the following additional difficulties:

Components of the Function Point Analysis Procedure and Mental Models of Software Functionality. The components of the function points procedure (including Inputs, Outputs, External interfaces, Queries and Files) do not readily match the object building blocks of the I-CASE development environment we studied. As a result, project managers reported to us in individual and group interviews that mapping HPS objects to function point analysis concepts was not easy. Function point analysis appears to require them to depart from the mental model they have for the user functionality and size of software. They also indicate that expending significant effort to examine the code within a module or an object still would result in a subjective classification of FUNCTION-COUNTs and potentially lower consistency in resulting function point estimates by different analysts.

Complexity Weights in Function Point Analysis. Classification of FUNCTION-COUNTs into simple, average and complex levels of complexity according to the function points methodology is also problematic (BANK91). 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 HPS development the ratio between the time required to code a simple type and a complex type may not be as large as it was in traditional development environments.

One might conclude from this that the complexity classification dimensions and weights used in the function point analysis method may not do as well in estimating the actual level of software development labor consumed. The weights applied to the different complexity levels were determined by Albrecht by trial and error (ALBR83). Moreover, Symons (SYMO88) recently suggested that a new set of weights might need to be recalibrated for any new technology, organization or development atmosphere. Clearly I-CASE qualifies as a technology that might require Albrecht's weights to be recalibrated.

[2]Computerworld recently reported on research results obtained by Chris Kemerer of the Sloan School of Management at MIT that showed that function point estimates of software size, regardless of the specific technique employed to obtain them, tend to vary little more than plus or minus 10%. This suggests their robustness for comparative measurement purposes.

Automatically Generated Code. A major source of the power of I-CASE tools comes from their ability to generate code during the testing and implementation phase. Yet a programmer or an analyst who has not written the actual code and done only the logical design would be forced to deal with the automatically generated code. The code is unlikely to closely match what a person would write. Thus, analyzing I-CASE-generated code would be an onerous and, most likely, an inefficient task. The alternative is to analyze object-based documentation directly.

Impact of Code Reuse on Labor Estimates in Construction, Testing and Implementation. The I-CASE methodology used in HPS development enforces modularization of application code and object-based design, which both promote more efficient system development and maintainability. Straightforward identification of FUNCTION-COUNTs is prone to double-counting of the labor consumed, since functionality derived from code reuse would consume comparatively little labor. The presence of a *central repository* in HPS plays a significant role. Although counting the five function types (or the number of objects, for that matter) provides a ballpark reading on the size of the product, function point-based estimates of labor will need to be adjusted for the leverage that code reuse creates.

In the I-CASE environment we have been studying, reuse seems to affect effort expended during the construction phase far more than any other factor. Yet reuse also contributes 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.

Code Reuse: A Measurement Void in Software Metrics

The preceding discussion suggests the importance of measuring the impacts of code reuse to improve labor estimation and I-CASE productivity assessment. Intuitively, 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). Yet as Hall (HALL87) and others (BANK90C) have shown, this intuitive definition does not really offer a perspective that helps managers to understand the extent to which code reuse leverages development productivity, software costs or software functionality in a meaningful way.

There are few rigorous definitions of code reuse as it applies to software development performance evaluation (LANE84, NUNA89, RAJ89). This measurement void is explained by the fact that prior research on code reuse in traditional development environments concentrated on the problems of encouraging it; they did not attempt to measure it. Another issue is just what a "code reuse metric" actually measures. Standish (STAN84), for example, argued that reuse should be measured at the line of code level in 3GL environments. But, this approach does not make sense for machine-generated code associated with CASE development. Generated code is likely to be verbose and bear little resemblance to what a human developer would produce in similar circumstances. Another problem is the extent to which this view of code reuse fits other aspects of CASE development, especially in terms of the conceptual model of a system. Neighbors (NEIG84) argued in favor of abstracting from source code to represent reuse in a meta-language. We find this view to be more useful in conceptualizing "code reuse" in I-CASE environments such as HPS or IEF. Both share a common feature: a high-level representation of an application that uses "information engineering" concepts as its basis (TI90).

4. OBJECT-BASED SOFTWARE METRICS FOR INTEGRATED CASE

Object-based I-CASE environments offer interesting opportunities to examine new metrics for measuring software development performance. For example, HPS offers support for object-based software metrics through its storage in a central repository of all objects enabling a historical record of application development to be maintained, and an abstract object hierarchy that defines the functionality of an HPS application. (Our prior research has shown that the structure of this hierarchy can be exploited to support automated function point analysis.)

Object-Based Development in HPS

The central repository used in HPS stores information about different kinds of objects used in applications developed with the tool. Examples of object types defined for use in HPS 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 a 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 HPS.

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. (See our other work for detailed illustrations and sample object hierarchies for typical investment banking systems (BANK90A and BANK90C).)

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.

Object-Based Code Reuse Metrics for I-CASE

Creating code reuse metrics to match the needs of object-based environments like HPS is a natural next step. We have suggested elsewhere that code reuse be measured in terms of managerial dimensions that identify the scope, value and effects of this practice. This resulted in a proposal for three classes of code metrics: *reuse leverage*, *reuse value*, and *reuse classification* (See BANK90C for additional details on these metrics, and an illustration of their application to a representative investment banking system.). The metrics we proposed rely on an analyst being able to identify objects that are reused, rather than lines of codes or modules included in an application.

Reuse leverage measures the number of times that an object is used within a system. This provides a very rough estimate (unadjusted for labor) of the extent to which developers "leverage" the contents of the repository. More formally, reuse leverage within a system is defined as:

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

This measure can be used at several levels of analysis, for example, in aggregate form for all the objects in an application, or by object type within the repository such as RULE SETs, SCREEN DEFINITIONs or USER REPORTs. The primary value of examining reuse leverage is that it enables an analyst to identify what is being reused and how much reuse is occurring.

To measure the productivity gains associated with code reuse in terms of their value to the firm, we must distinguish between the reuse of easily-programmed objects and the reuse of more costly objects. We compute *reuse value* by weighting the level of reuse by the cost of programming the various types of objects that are reused. A more formal definition of reuse value is:

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

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.*

This gives the analyst a reading on the relative savings obtained from reuse.

We normally include in our computation of code reuse for a project any object which is found in the repository, rather than just those that are written from scratch. But for some managerial purposes, it may be useful to *classify* reuse according to its origin, how much it occurs in work by project teams or individuals or over time, and whether it extends beyond the boundaries of an application or a project. Two important categories of reuse are internal and external reuse. *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 that are in the repository, but where those objects "belong" to and were originally developed for a different system. Though 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. The latter is desirable since observing its occurrence indicates that developers are leveraging the contents of the I-CASE repository.

Object-Based Functionality Metrics for I-CASE

The function point analysis approach can be adapted to support the measurement of software outputs in object-based systems more directly. We have explored two object-based functionality metrics. The first, *OBJECT-COUNTs*, is determined by summing the occurrences of individual objects types in an application. (It is similar in spirit to FUNCTION-COUNTs as shown in the definition of function points in Sidebar 2.) The second, *OBJECT-POINTs*, is defined as follows:

$$\sum_{w=1}^{3} \sum_{t=1}^{4} OBJECT\text{-}EFFORT\text{-}WEIGHT_{wt} * OBJECT\text{-}OCCURRENCE_{wt}$$

where

OBJECT-EFFORT-WEIGHT$_{wt}$	=	*average estimated development effort associated with object type t of complexity w (LOW, AVERAGE, HIGH), based on project manager heuristics;*
OBJECT-OCCURRENCE$_{wt}$	=	*number of occurrences of one of several object types t of complexity w (including RULE SET, 3GL MODULE, SCREEN DEFINITION and USER REPORT and others) in an HPS application.*

The reader should note that OBJECT-POINTs incorporate information that distinguishes among the levels of complexity for each object type in terms of the labor required, rather than with summary effort weights that only distinguish between object types. Exploratory research we have conducted showed that OBJECT-COUNTs provide estimates of software size that are well-correlated with the labor required to produce them (BANK90C). Additional research on this issue is in process at our research site, and we are extending it to incorporate a more detailed effort weighting scheme based on project manager heuristics (KUMA91).

5. AUTOMATED SOFTWARE METRICS IN INTEGRATED CASE

The metrics we have propose are not meant to be collected manually, although it may be reasonable for managers to make preliminary estimates of the number of objects a system will contain, or the extent to which external reuse is possible. Instead, we advocate the use of specially-developed automated analysis tools that are as much a part of the I-CASE development environment as the rapid screen and report painters, and automated documentation-generation and software testing facilities are a part of the CASE tool. We now consider the means and rationale for automating function point and code reuse analysis for object-based I-CASE.

Automated Analysis in the HPS Environment

Automating software metrics in HPS begins with a major advantage over similar efforts in 3GL environments. For example, the hierarchical application meta-model that is stored by HPS can be used to exhaustively identify objects associated with any application system. Following the chain of relationships between objects enables an "automated analyst" to identify all the objects accessed or invoked by a given object. As a result, much of information needed to calculate function points, OBJECT-POINTs, and code reuse exists 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.

Another important feature of the I-CASE tool we studied is the HPS development language. It permits a precise mapping to be made between every 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.)

Automating Function Point Analysis

Using this approach we designed an architecture for an *automated function point analyzer* for HPS with our colleagues Eric Fisher, Charles Wright and Dani Zweig (BANK90C). The architecture has three main components -- an Object Identifier, a Function Counter and a Complexity Factor Counter -- and is shown in Figure 1.

INSERT FIGURE 1 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 a mapping from objects and entity relationships, to function types and complexities, and finally to FUNCTION-COUNTs. The *Complexity Factor Counter* computes environmental complexity, used in function point analysis as an adjustment factor to allow for the overall complexity of the task and the environment in which it is implemented. The scores for fourteen complexity factors are computed 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.

Automating Code Reuse Analysis

Karimi (KARI90) has observed that unmanaged reuse of code in CASE environments is likely to result in suboptimal development performance. An object-based I-CASE environment like HPS also provides a major assist for the implementation and control of code reuse. HPS 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-based system may be designed so that each such routine is a unique object. This makes reuse opportunities considerably easier to identify and to exploit.

HPS also serves to support the control of code reuse, and this increases the value managers will attach to measurement. With the design of the entire system stored centrally along with the software itself, instances of code reuse can be identified as multiple calls to an object within the repository.

To follow up on these ideas, we designed an *automated code reuse analyzer* for use within HPS. 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. 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, such as the ones we discussed earlier, can be computed.

Automating Object Analysis

The *object point analyzer* operates along the same lines. By-products of automating the analysis of an HPS 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 1) is instantiated based on the function point analyzer's scan of the application meta-model stored in the central repository. This in turn results in information about the typed OBJECT-COUNTs for an application. Additional functionality is required to represent the relative levels of effort required to build objects of different levels of complexity. This information can be obtained through empirical analysis of the characteristics of repository objects of different levels of complexity and the time expended to build them. Then, the results of this analysis can be stored in a table that is accessible to the object point analyzer, so that it can adjust OBJECT-COUNTs to arrive at a measurement of OBJECT-POINTs (BANK91).

6. AUTOMATED REPOSITORY ANALYSIS FOR SOFTWARE ASSET MANAGEMENT

Automating the collection of software metrics provides management with new tools to manage software development projects, as well as the "software assets" of the firm. A 1980s definition of the term "software assets" would involve identifying the sum total of the code, routines and code libraries, and databases that deliver the functionality and content of 3GL software applications. The value of such applications would be largely derived from the extent to which the applications improved cost control or helped to generate revenues -- in the firm's business activities.

It would be hard, however, to value an installed base of 3GL code in terms of the leverage created for future development. The same is *not* true for I-CASE, however. I-CASE opens up opportunities for code reuse, which means that repository software becomes a "leveragable" asset. We now turn to a discussion of automated software project tracking, repository evaluation and our proposal for a new software asset management perspective for I-CASE.

Automated Software Project Tracking

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 function point and productivity analysis had to be performed. But, point estimates only described *the outcome* of development activities; they failed to describe *the process* leading to the delivery of completed software. Yet this process is what management needs to fine-tune so that the outcome can be improved.

With object-based I-CASE, it is possible at the project level for management to replace point estimates of productivity with a full software development life cycle *trajectory* of performance estimates by using automated software metrics (BANK90F). 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 objects that are required to achieve such functionality. Order of magnitude estimates can be made 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 it may not be entirely built.

Automated software metrics will be even more useful as a project moves into the construction and testing phases. Figure 2 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. (We assume that the capability exists for the automated analysis tools to access data in a separate accounting system that tracks the billable hours developers spend on projects.)

INSERT FIGURE 2 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 2, 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 2 provides additional illustrative 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 design and code reuse capabilities of the CASE tool), they nevertheless suggest the possibility to management to take additional steps to manage code reuse to avoid substandard development productivity results.

Automated Repository Evaluation

Although the example we have used is highly simplified, other useful comparisons of descriptive project performance metrics based on *automated repository evaluation* would pave the way for management to obtain a fuller understanding of the dynamics of software development. For example:

- reuse classification metrics can be used to compare the relative productivity gains obtained from internal versus external reuse, since as the repository matures, the leverage created by external reuse should increase;
- 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 an I-CASE tool matures and new capabilities become available;
- 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 gauge the effects of other variables including team size, experience levels, developer training and the size of an application.

Tracking the software development life cycle of an I-CASE application with automated performance trajectory metrics offers management the chance to obtain a more comprehensive understanding of a firm's software development operations. As our sketch of the function point analyzer suggested (see Figure 1), 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 costs of large scale software development, and offer the firm a new range of opportunities to achieve competitive advantage.

However, repository analysis need not be confined to the project meta-model hierarchy level. The real potential of this approach is to examine the contents of the entire repository in terms of its objects or object-based modules it stores. Analysis of "repository demographics" will enable a new range of questions about code reuse to be answered by senior managers. For example:

- Is reuse biased by object type? Module type? Mostly limited to a programmer's reuse of his own or his project's objects? Does it occur more within project teams than across them?
- How does reuse change over time? Does it increase as the number of objects in the repository

increases? Or, is increasing reuse related to the maturation of the CASE toolset?

* Is there a small subset of objects that receive the highest levels of reuse? What are they like? How are they reused? Are there other kinds of objects that one might expect to be included in this subset that are not members? If so, why are they missing?

* Do "expert" programmers or relative newcomers exhibit the highest levels of reuse? Is this related to the training they receive on the tool or their knowledge of the repository?

* On a repository-wide basis, what levels of reuse are exhibited? Overall object reuse leverage? Reuse value?

In addition to code reuse, other issues can be studied directly as well. Automated repository evaluation can yield information on the relative frequency with which different kinds of objects are created. Such analysis can help management to focus on those aspects of the I-CASE environment that need special attention. This could lead to the creation of more powerful facilities to assist with the design and construction of these objects. It would also be possible to examine whether object size is growing or shrinking on average, reflecting system analysis, design and development practices in a more mature I-CASE environment.

The Software Asset Management Perspective

When this kind of information is obtained, senior management will be in a position to estimate the overall leverage that CASE creates in achieving cost reduction and boosting software functionality. Perhaps even more important for the long haul though is that management understand the qualities of the firm's software development environment. In our recent repository evaluation study with Dani Zweig (BANK90E), we proposed three kinds of potential bounds on development performance that management can address based on repository demographics:

*(1) There will be a **technical bound** on development productivity that is attributable to the contents of the CASE tool set.*

Most development managers are acutely aware of the strengths and weaknesses of the technical development environment. At the First Boston Corporation we have obtained evidence that the firm's CASE facilities do well in promoting code reuse. (See BANK90A and BANK90E for additional details.) But additional repository analysis evidence suggested that the company probably has not reached its *technical bound* on development productivity. Repository evaluation can be used to gauge the extent to which new CASE tool features to be implemented in the future can raise this bound.

*(2) There will also be an **organizational bound** on development productivity that can be explained in terms of how well software development management is able to motivate developers to achieve high levels of*

reuse.

Our research has shown that there are many issues that will need to be resolved to lift the organizational bound on development performance. For example, at First Boston we observed conflicting interests between the builder of a given object and a potential reuser of that object. The problem is that the builder of the object also is required to be the "guarantor" of the object's performance when it is reused. Since this may require effort on the part of the original developer (when he perceives his work on the object already to be complete), there may be incentives for the reuser to revise the object in a way that meets the need, without directly involving the original developer. (We call this practice "hidden reuse.")

One potential organizational solution that management should consider to deal with the "*object ownership*" problem is to transfer all objects to a neutral third party when the project requiring its construction reaches completion. This could be an "object administrator" (parallelling the role of a "database administrator" for an I-CASE environment), who is charged with identifying and promoting firm-wide reuse of a *core set of reusable objects.* If this core set of reusable objects contains the right building blocks, and management promotes reuse well through training, monitoring and incentives, the leverage on development productivity that can be created by the repository should increase. This, in turn, will increase the overall business value of the firm's software assets.

*(3) There also will be an **architectural bound** on the software project development process, and this bound may have the most important implications for the long-term.*

Repository evaluation can also shed some light on the architectural aspects of I-CASE development. The basic questions of interest here are: How is development carried out? For example, how large are the typical applications developed with I-CASE? What guidelines do developers follow in programming to achieve reuse? What happens in the early life cycle phases? Is the firm's architectural perspective on software development that high levels of reuse should be tied to activities conducted in the analysis and design phases, and not just construction? What levels of code reuse can be targeted? What minimum levels should be mandated? How will they change over time?

Each of these questions can be understood better by management when evidence is available from repository evaluation. With answers to some of these questions, management can then fine-tune development activities to minimize the costs of project development, while maximizing the business value of the software assets stored in the repository to support future development activities.

REFERENCES

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. Forthcoming in *MIS Quarterly.*

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. In the *Proceedings of the Twenty-third Annual Hawaii International Conference in Systems Science*, January 1991.

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.

BANK90E Banker, R. D., Kauffman, R. J. and Zweig, D. Factors Affecting Code Reuse. Working paper, Stern School of Business, New York University (December 1990).

BANK90F Banker, R. D., Kauffman, R. J., and Kumar, R. Managing the Performance of Computer Aided Software Engineering (CASE) Development with Dynamic Life Cycle Trajectory Metrics. Working paper, Stern School of Business, New York University (October 1990).

BANK91 Banker, R. D., Kauffman, R. J., and Kumar, R. An Approach to Constructing Productivity Assessment Metrics in Computer Aided Software Engineering (CASE) Environments. Working paper, Stern School of Business, New York University (April 1991).

BOOC89 Booch, G. What Is and What Isn't Object-Oriented Design. *Ed Yourdon's Software Journal*, 2(7-8), pp. 14-21, Summer 1989.

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.

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.

KARI90 Karimi, J. An Asset-Based Systems Development Approach to Software Reusability. *MIS Quarterly*, June 1990, pp. 179-198.

KUMA91 Kumar, R. *Development Effort Estimation in Integrated Computer Aided Software Engineering Environments: An Object-Centered Approach.* Doctoral dissertation, Stern School of Business, New York University, in progress.

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.

MEYE88 Meyer, B. *Object-Oriented Software Construction.* Prentice Hall, New York, NY, 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, Hawaii (January 1989), pp. 959-968.

POLL90 Pollack, A. The Move to Modular Software. *New York Times* (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.

TI90 Texas Instruments. *A Guide to Information Engineering Using the IEF: Computer Aided Planning, Analysis and Design, Second Edition.* Dallas, TX (1990).

Sidebar 1. High Productivity Systems (HPS) -- An Integrated CASE Environment

The First Boston Corporation, a large investment bank located in New York City made the initial commitment to design and develop an *object-based, repository-based integrated CASE (I-CASE) environment called "High Productivity Systems"* (**HPS**) at a cost of nearly $90 million over the course of three years. HPS was built by the firm as a response to the problems it faced in developing and maintaining technically complex systems. 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 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 "cooperative processing" systems that run simultaneously on all three platforms, and reduce the firm's reliance on three sets of highly skilled and costly programmers.

HPS applications are written in an *object-based* language (BOOC89) 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. (Note that we use the term "object-based" to distinguish this environment from others that involve "object-oriented" concepts, where the inheritance properties of objects are central.

A special feature of HPS is its *object repository*. 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.

Sidebar 2. Function Point Analysis: An Overview

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:

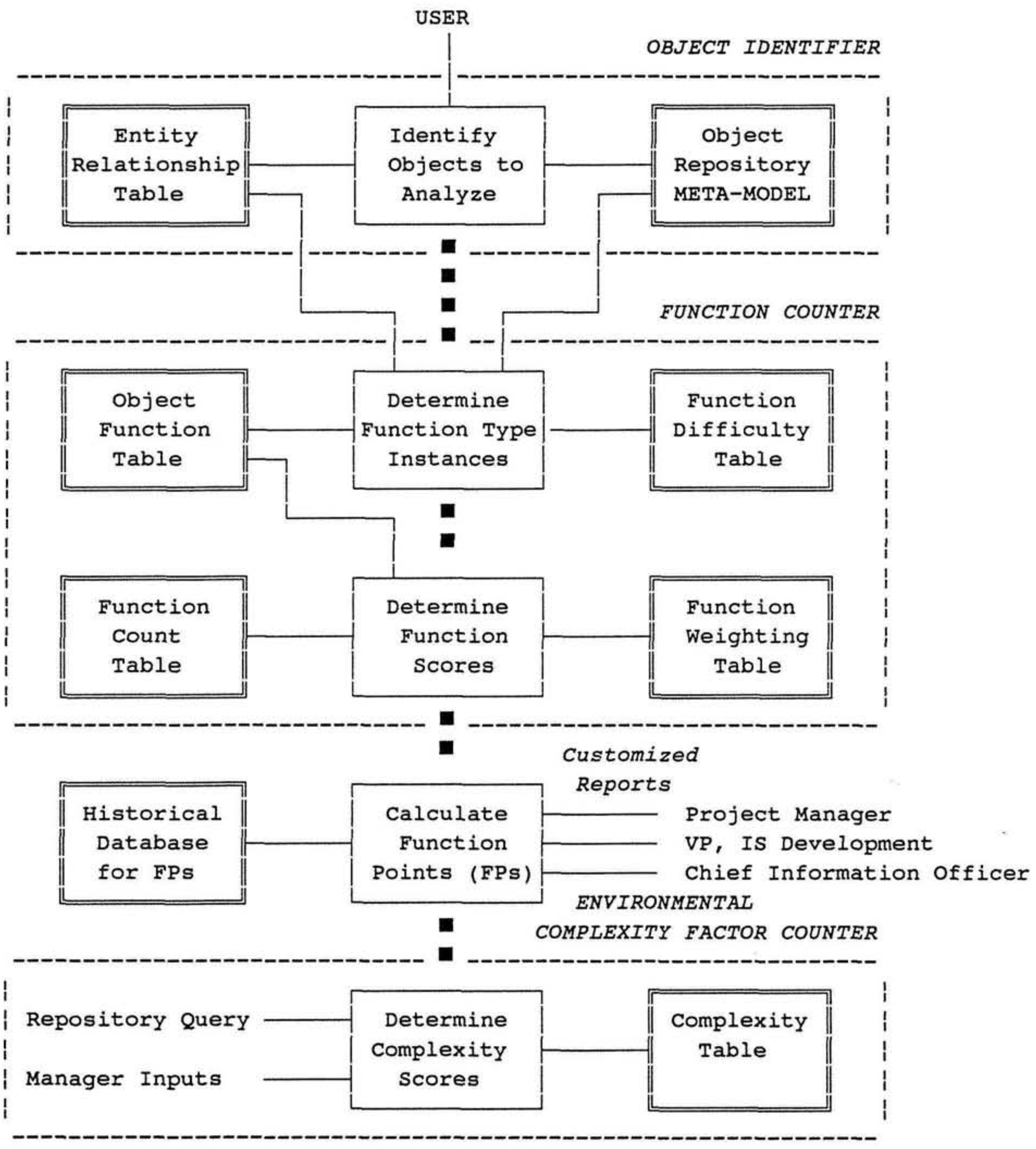
$$FUNCTION\ POINTS = FUNCTION\ COUNT * (.65 + (.01 * \sum_{f=1}^{14} COMPLEXITY_f))$$

where

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

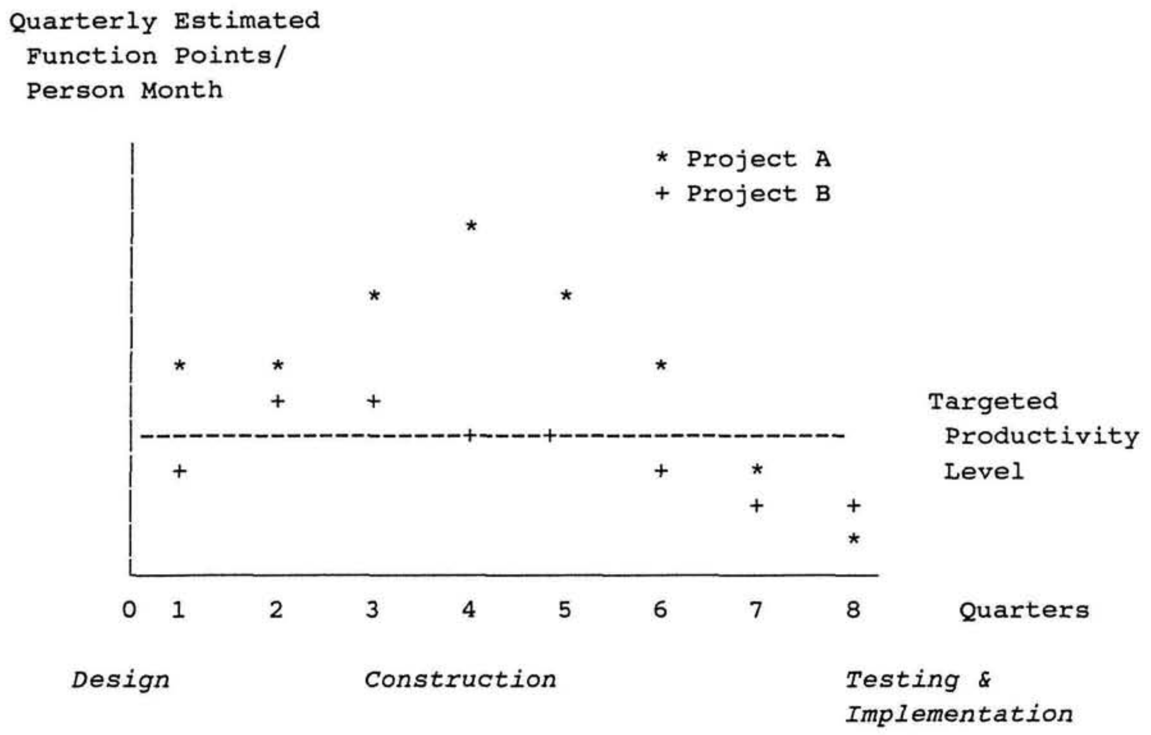
Function points are meant to provide a language-independent and implementation-independent measure of the functionality actually produced and delivered to the user. They differ from output measures, such as those based on source lines of code, that focus on how much code constitutes an application. Thus, function points are not sensitive to the level of the language in which an application is written. Since its introduction in the late 1970s, function point analysis has evolved into a well-accepted and operationally well-defined methodology.

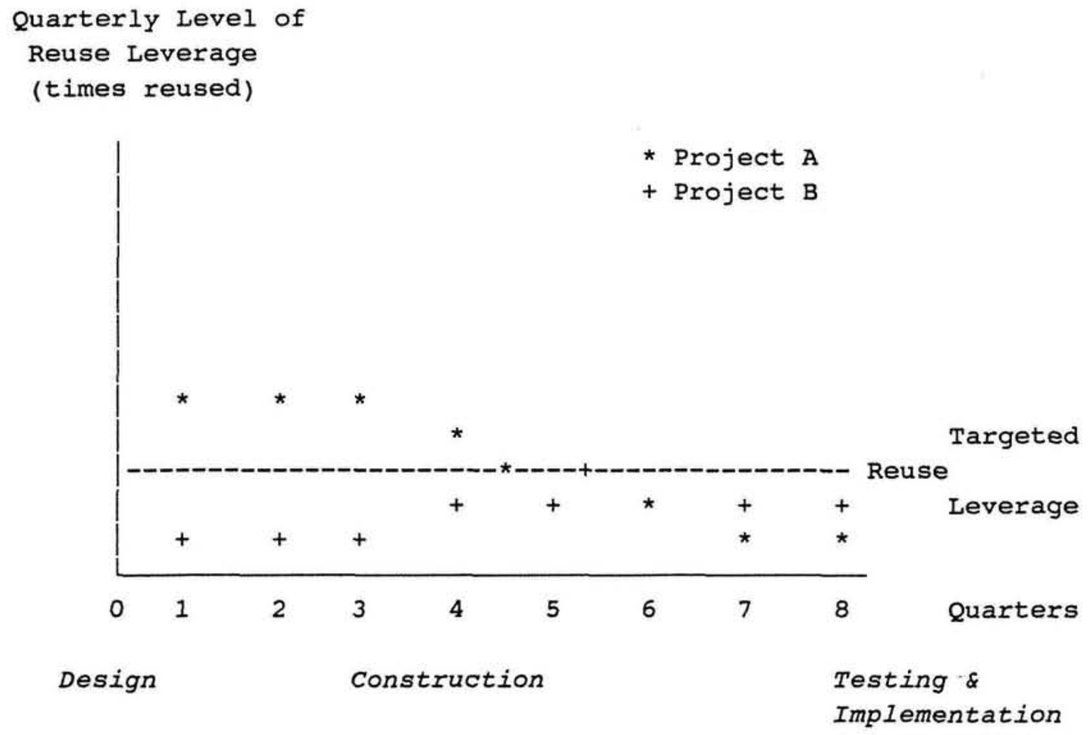
Figure 1. The Automated Function Point Analyzer: A Schematic


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
ENVIRONMENTAL COMPLEXITY FACTOR COUNTER
Repository Query
Manager Inputs
Determine Complexity Scores
Complexity Table

The *function point analyzer* uses the HPS application meta-model to *identify objects.* It then *assigns FUNCTION-COUNT scores* to those objects and *weights them according to their complexity.* The final step further revises the FUNCTION-COUNTs based on the environmental complexity of the development effort. This requires programmer or manager input in parallel with repository queries.

Figure 2. Software Development Performance Trajectories: Function Points and Reuse


Quarterly Estimated
Function Points/
Person Month
* Project A
+ Project B
Targeted
Productivity
Level
Quarters
Design
Construction
Testing &
Implementation


Quarterly Level of
Reuse Leverage
(times reused)
* Project A
+ Project B
Targeted
Reuse
Leverage
Quarters
Design
Construction
Testing &
Implementation