# AUTOMATING OUTPUT SIZE AND REUSE METRICS IN A REPOSITORY-BASED COMPUTER AIDED SOFTWARE ENGINEERING (CASE) ENVIRONMENT

Rajiv D. Banker

Robert J. Kauffman

Charles Wright

Dani Zweig

Department of Information, Operations, and Management Sciences

Leonard N. Stern School of Business, New York University

44 West 4th Street, New York, NY 10012

# AUTOMATING OUTPUT SIZE AND REUSE METRICS IN A REPOSITORY-BASED COMPUTER AIDED SOFTWARE ENGINEERING (CASE) ENVIRONMENT

**Rajiv D. Banker**
Carlson School of Management
University of Minnesota

**Robert J. Kauffman**
Stern School of Business
New York University

**Charles Wright**
Financial/Brokerage Systems
Seer Technologies

**Dani Zweig**
U. S. Naval Post Graduate School

**Working Paper Series**
STERN IS-93-39

# AUTOMATING OUTPUT SIZE AND REUSE METRICS

# IN A REPOSITORY-BASED

# COMPUTER AIDED SOFTWARE ENGINEERING (CASE) ENVIRONMENT

November 19, 1992

**RAJIV D. BANKER**
Arthur Andersen/Duane R. Kullberg Chair in Accounting and Information Systems,
Carlson School of Management, University of Minnesota
Minneapolis, MN 55455
Phone: 612-624-2540
Fax: 612-626-7795
E-mail: rbanker@csom.umn.edu

**ROBERT J. KAUFFMAN**
Associate Professor of Information Systems
Stern School of Business
New York University
44 West 4th Street
New York, NY 10012
Phone: 212-998-0824
Fax: 212-995-4228
E-mail: rkauffma@stern.nyu.edu

**CHARLES WRIGHT**
Senior Consultant, Financial/Brokerage Systems
Seer Technologies
5 Penn Plaza, 14th Floor
New York, NY 10001
Phone: 212-643-6000
Fax: 212-643-6146

**DANI ZWEIG**
Assistant Professor of Information Systems
U.S. Naval Post Graduate School
Code AS/Zg
Department of Administrative Sciences
Ingersoll Hall
Monterey, CA 93943-5000
Phone: 408-646-3439
Fax: 408-646-3407
E-mail: dani@netcom.com

*[KEYWORDS: CASE, computer aided software engineering, function point analysis, object-based development, programming productivity, repositories, reuse, software costs, software engineering economics, software metrics.]*

**AUTOMATING OUTPUT SIZE AND SOFTWARE REUSE METRICS**

**IN A REPOSITORY-BASED**

**COMPUTER AIDED SOFTWARE ENGINEERING (CASE) ENVIRONMENT**

---


## ACKNOWLEDGEMENTS

We wish to acknowledge Mark Baric, Gene Bedell, Tom Lewis and Vivek Wadhwa for the access they provided us to the software development activities and staff at The First Boston Corporation and Seer Technologies. We also appreciated the assistance and advice of Donna Dodson, Len Erlihk, Gig Graham, Don Middleton, Michael Oara, Norman Shing and Brian Weisinger, and research assistance of Eric Fisher and Vannevar Yu. We thank Hank Lucas for valuable suggestions on the presentation of the ideas in this paper. Robert J. Kauffman also thanks the Nippon Electric Corporation for partial funding. The research presented in this paper enabled the development of *Project Metrix*, an automated function point and software reuse analysis facility built at Seer Technologies in conjunction with its integrated CASE tool, *High Productivity Systems* (**HPS**).


---

# AUTOMATING OUTPUT SIZE AND SOFTWARE REUSE METRICS IN A REPOSITORY-BASED COMPUTER AIDED SOFTWARE ENGINEERING (CASE) ENVIRONMENT


## ABSTRACT

Measurement of software development productivity is needed in order to control software costs, but it is discouragingly labor-intensive and expensive. Computer aided software engineering (**CASE**) technologies -- especially repository-based, integrated CASE -- have the potential to support the automation of this measurement. In this paper, we discuss the development of automated analyzers for function point and software reuse measurement for object-based CASE. Both analyzers take advantage of the existence of a representation of the application system that is stored within an object repository, and that contains the necessary information about the application system. We also discuss metrics for software reuse measurement, including *reuse leverage*, *reuse value* and *reuse classification*, that are motivated by managerial requirements and the efforts, within industry and the IEEE, to standardize measurement. The functionality and the analytical capabilities of state-of-the-art automated software metrics analyzers are illustrated in the context of an investment banking industry application, that is similar to systems deployed at the New York City-based investment bank where these tools were developed and tested.

# 1. INTRODUCTION

## 1.1. The Incentive and Opportunity to Automate Software Metrics

The recent upsurge in interest concerning computer aided software engineering (**CASE**) technologies [59] provides managers with both an incentive and an opportunity to measure software development performance. The incentive is that documenting the productivity gains from CASE can help to justify (or, for some products, discourage) the large investment the technology often requires. One popular press observer of these developments has recently written:

> *"Like handcrafted furniture, software has traditionally been customized for a task in a laborious process more akin to artistic work than to engineering. [But now], software is increasingly being written in the form of pre-fabricated pieces that can be reused in different combinations, much as plumbing systems can be tailored for each house yet still be built out of standard pipes, valves and joints."* ([56], pp. D1-2)

Many observers believe this is a "software industrial revolution" in the making, a view that has been held in the computer science research community since the 1970s [38, 54]. However, the cost of participating in this revolution may be substantial, while the benefits have proven hard to verify [12, 47, 48].

The opportunity is that of automating the collection of productivity data. Any firm with high software expenditures has a strong incentive to control and improve its software development productivity, and this requires measurement [17, 27, 39, 43, 50, 61]. But in traditional software shops, such measurement requires discouragingly expensive manual analysis of the software. CASE technologies, especially repository-based integrated CASE technologies, provide a means to automate a variety of software metrics

that can help managers to gain better control of their software development operations.[1]

Automation of the process of collecting key software metrics is likely to be one of the next areas to receive attention from CASE tool vendors. *Software Magazine* expressed a similar view of the future by showcasing products from nearly forty vendors that measure productivity within a CASE environment [12]. Very few of these, however, automate the collection of the software metrics needed for productivity analysis. The majority are project management tools which require a significant amount of input from the user to make them useful. The magnitude of this manual burden is precisely what has made productivity measurement so difficult to carry out in the past.

In this paper we examine the automation of two important metrics: *function points* -- a measure of programmer output in terms of software functionality -- and *software reuse* -- a major determinant of programmer productivity. Function point analysis is a widely-accepted means of measuring output in MIS software development, but it is very labor intensive, especially for large systems, and this has limited its adoption.

Software reuse allows organizations to take advantage of previous development efforts, rather than paying to create every system from scratch. Extensive reuse in the construction phase can increase productivity by an order of magnitude and more [4], due to the use and invocation of previously developed software modules. But the reader should recognize that reuse offers a philosophy for software development that extends to *every* phase -- to include reuse of abstract representations of a system [42], software objects [44, 45] and reusable components [46], prototypes and partial systems [57], data and data models, program architecture and data structure designs [20, 37], and downstream life cycle processes (such as implementation and test routines) [60]. In this way, software reuse offers the potential to create even greater long-term benefits, especially when efforts to reuse extend to include early life cycle planning

[1]For an introduction to the "repository" concept, see [23], [29], [36] and [43].

activities, enabling development of systems that share common architecture and common design elements [59]. In fact, it has been recognized that it is highly desirable to conduct software development projects that result in reusable objects, that can be then used widely by various development projects within a firm. Reuse, other than the explicit invocation of previously written modules, has proven difficult to identify, let alone measure. Software reuse analysis, like function point analysis, requires knowledge of the semantics of the software being analyzed.

This paper focuses upon function-point and software-reuse measurement in the construction stage of software development. As we shall see, however, this measurement supports *ex ante* cost estimation for the entire development process, as well as providing *ex post* insights into the level of productivity achieved in CASE development environments. To automate the computation of these metrics, we require the ability to automate the analysis of the *content* of the software being analyzed. We shall see that, in addition to other benefits claimed for it, *repository object-based development* can provide this capability, primarily by encouraging the division of software into more easily analyzed units than the traditional procedure-oriented program.

A prerequisite for gauging the strength of any "industrial revolution in the making" is the ability to measure such basic factors as output and productivity. Despite annual software costs rising into the hundreds of billions of dollars, and a general agreement that these costs must be controlled [8, 9], such measurement has proven too difficult and expensive for most organizations. We will examine the potential of modern software development tools to not only increase the productivity of the software development function, but to finally begin to provide management with an understanding of how to bring it under control.

### 1.2. Organization of the Paper

In this paper, we will describe the design and common architecture, and managerial application of two

automated software metrics analyzers made possible using a *repository-based Integrated CASE Environment* (**ICE**). These include a *Function Point Analyzer* (**FPA**) and a *Software Reuse Analyzer* (**SRA**). The remainder of the paper is organized as follows. Section 2 introduces the basic concepts necessary to understand our strategy for developing the automated software metrics facilities. It includes: an overview of the function point analysis methodology; a discussion of why the methodology is useful, but costly and problematic to implement; a consideration of prior attempts to automate function point analysis; and an examination of the features of repository object-based CASE development environments that enable us to automate function point analysis. Section 3 presents the details of the Function Point Analyzer. We make the argument that much of the necessary information for a function point analysis is readily available in an application's meta-model, and we show how the repository objects and the relationships between them can be mapped into function point analysis.[2] We present the architecture for FPA and then illustrate how it navigates the hierarchy of rules to conduct an exhaustive search of the user functionality built into an application.

Section 4 presents the Software Reuse Analyzer. We discuss three classes of software reuse metrics that are prompted by recent efforts to standardize such measurement, explain the design of SRA, and describe the manner in which it navigates the application meta-model hierarchy to obtain the relevant information to instantiate the metrics. The concluding section addresses additional technical and managerial questions that were raised by our work in this area, and the future research required to resolve them. It also summarizes the key contributions of this work to practitioners and to research on software development productivity. The paper includes a stand-alone example of how the analyzers and the reuse metrics can be applied to an investment banking application called the *Broker Sales Reporting System*.

---

[2]The term "meta-model" builds on the idea of "meta-data," i.e., those elements of a data dictionary that describe "the keys, attribute order, formats, and rules applied to individual records and attributes in a database. A repository stores additional meta-data concerning many other aspects of the total system of which the database is only a part ([23], p. 47). In this paper, we focus almost exclusively on the capability of a repository to store information concerning the relationship between objects which comprise a system.

## 2. AUTOMATING FUNCTION POINT ANALYSIS: PRELIMINARIES

### 2.1. Function Point Analysis

The magnitude of a software development project's 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 productivity factors ranging from the quality of the hardware used by the programmers to the sophistication of the users requesting the software [64]. *Function point analysis*, originally developed by Allan Albrecht of IBM, provides a summary measure of the functionality of a system, and is especially useful as a descriptor of MIS applications. This measure, modified by a measure that incorporates the influence of environmental productivity factors, provides an empirically tested basis for managers to estimate the resources required to build systems of various sizes [1, 2].

Function points are meant to provide a language-independent and implementation-independent measure of the functionality actually produced and delivered to the user. In this, they differ from code-output measures (such as source lines of code) that can reward verbose programming practices. Since its introduction in the late 1970s function point analysis has evolved, with the help of the International Function Point Users Group (**IFPUG**), into a well-accepted and operationally well-defined methodology that is used in many firms [18, 61].[3]

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. *Function counts* are determined by computing a

---

[3]For additional details on the implementation of function points which extends the approaches presented by Albrecht and Gaffney [2] and Zwanzig [70], see Symons [63], who discusses function points with entity type complexity rules.

weighted sum of the point scores which are assigned (on the basis of their complexity) to each External Input, External Output, Logical Internal File, External Interface and Query that comprise the system. The weights depend in part upon the complexity of the given inputs, outputs, etc., as determined by the number of data elements and relations involved. Function counts are further adjusted by a measure of the *environmental complexity* when a project is implemented. The mathematical definition of *function points* is shown below.

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

where

*FUNCTION-COUNTS* = *the sum of the instances of the five function types, including External Inputs, External Outputs, Logical Internal Files, External Interfaces and Queries;*

$COMPLEXITY\text{-}FACTOR_f$ = *a variable, f, associated with one of fourteen descriptors of the implementation complexity of a system.*

Two papers provide useful critiques of function point analysis, alternative definitions and the issues that arise in calculating and using them in practice [34, 64]. (Appendix 1 offers a more in-depth description of the mechanics of function point analysis, and includes a summary of the fourteen complexity factors.)

One roadblock to collecting function point metrics for software applications is that their computation, usually performed manually, is very labor-intensive. In addition, such computation requires the availability of consistently good system documentation. In practice, where design documentation exists at all, it too often describes the system as it was originally designed, rather than as it was finally delivered. This can

force the analyst to spend even more time analyzing the code to determine the extent to which the design documentation reflects the functionality that was actually produced.

A third concern is that of *calibrating* the people who carry out the function point analysis. Our experience in a study of the productivity of CASE development suggested that even when well-trained individuals perform function point analysis for the same set of software projects, there are bound to be discrepancies which have to be resolved [4]. Individual differences in interpretation of documentation, knowledge of an application and experience in conducting function point analysis can all drive these differences. Low and Jeffrey [40] examined the reliability of function point analysis in a more structured manner and found that significant training in the use of the complexity measures is necessary to ensure that the correct constructs are being measured. More recently, Kemerer [34] found evidence to support a more optimistic view: his empirical work showed that counts differ no more than about plus or minus 10% across well-trained analysts. This level of agreement, again, requires a substantial manpower investment, first in training and subsequently in analysis.

### 2.2. ICE -- A Repository Object-Based Integrated CASE Environment

A large New York City-based investment bank made the initial commitment to design and develop a repository object-based integrated CASE environment 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 were geographically distributed, and were required to perform effectively on a 24-hour basis.

Similar to others in the investment banking industry, the firm had been experiencing rapidly mounting software costs that were expected to further rise as its trading activities expand to provide global coverage. To achieve competitive performance in this environment required the firm's developers to program

applications which were shared by three hardware platforms (mainframe, minicomputer, and microcomputer), each programmed 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 programmers.

ICE applications are written in a 4GL which buffers programmers from the complexity of the firm's operating environment. ICE automatically translates the 4GL code into the languages appropriate for the target platforms, and communication protocols for cooperative processing across platforms are handled without programmer intervention. Project managers and software developers whom we interviewed commented that development in this environment, with the strong emphasis on software reuse, and with much of the coding effort automated, tends to shift effort from the construction phase, to the analysis and design phases.

ICE maintains a meta-model whose structure is derived from *entity-relationship modeling* [14], and ICE was especially constructed to support the development of cooperative processing applications. The code is organized according to objects that play specific roles in the functions delivered by the application, and the various software functions can be allocated across hardware platforms in the most appropriate manner. This organization is also what makes it practical to automate the analysis of the code for the computation of function points.

A feature of ICE, of special interest for the discussion which follows, 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. In addition to the stronger control it provides, the advantage associated with 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 be written once, no matter how many times it is used. Such reuse has the potential to decrease software development costs, and it forces

developers to more carefully "engineer" an information and information systems architecture which will form a solid base for the firm's business. The repository also makes the automation of software reuse measurement practical, since it maintains a record of each object and where it is used or reused.

### 2.3. Definitions of Basic ICE Objects

The ICE object repository stores information about the different kinds of entities or objects which form the basic building blocks of ICE-developed applications: BUSINESS PROCESSES, RULE SETS, 3GL MODULES, SCREEN DEFINITIONS, FILES, DATA VIEWS, DATA ELEMENTS, DATA DOMAINS, REPORTS and REPORT SECTIONS. It is useful to think of these objects as similar to corresponding 3GL constructs. For example, a RULE SET is analogous to a 3GL procedure, and a SCREEN DEFINITION can be thought of as a window that provides a user interface. At the same time, it is worthwhile to keep in mind that the object definitions in the ICE environment are deliberately precise and rigid, with the result that an analysis of the meta-model gives us a great deal of semantic information about the application system, without forcing us to analyze the actual code. We next consider each object type in more detail.

A RULE SET contains most of the instructions which observers unfamiliar with CASE tools would tend to think of as "the program". Most of the "traffic control" resides there: a RULE SET can *use* other RULE SETS or 3GL MODULES, invoke REPORTS, which in turn invoke REPORT SECTIONS, *access* FILES and *communicate with* SCREEN DEFINITIONS. (The 4GL used by ICE has a specialized set of verbs to describe the various interactions among object types.)

A 3GL MODULE is a pre-compiled procedure, originally written in a specific 3GL. Although the 4GL language used by ICE developers is very small and general, it provides those 10% of the data handling and computational capabilities which constitute over 90% of the functionality of an information system. It is

left to 3GL MODULES to implement more specialized capabilities. In investment banking operations, highly quantitative options pricing and other valuation procedures for derivative instruments exist on the shelf in optimized 3GL code at most firms. Such procedures are used intact, as 3GL MODULES, rather than recoded.

A SCREEN DEFINITION is the logical representation of an on-screen image. A RULE SET can *communicate with* a given SCREEN DEFINITION, meaning that data is passed back and forth between them. The user-interface capabilities of a SCREEN DEFINITION are built into ICE, and do not have to be considered by the developer. This tends to speed the development process for screens in ICE. By comparison, the creation of screens delivered by IBM 3270 terminals using traditional development methods is more labor-intensive by a full order of magnitude [4].

A DATA VIEW consists of a set of DATA ELEMENTS, data objects that have been defined in the object repository. A DATA VIEW can be thought of as a logical data record. The communication of all data between ICE objects is mediated by DATA VIEWS. For example, data is passed from a RULE SET's DATA VIEW to a SCREEN DEFINITION's DATA VIEW and back. Data for a 3GL MODULE or a REPORT must similarly be passed through a DATA VIEW.

A REPORT means much the same thing in ICE as it does in other development environments. More specifically, a REPORT is the internal logical representation of the physical report. REPORTS consist of one or more REPORT SECTIONS, each with its own layout.

Each of these ICE objects is reusable, and good practice in the context of ICE development is to reuse them as much as possible. Placing all of the objects associated with an application in the object repository has two intended effects. It prevents a programmer from circumventing the discipline of database and object management, and it makes all the objects of one application available for reuse by any other

application which is stored in the repository.[4]

### 2.4. From ICE Repository Objects to ICE Application Meta-Models

An ICE application system consists of ICE repository objects, such as RULE SETS and SCREEN DEFINITIONS, communicating with each other in a structured manner. This is illustrated in Figure 1.

---

INSERT FIGURE 1 ABOUT HERE

---

A single application is invoked by a menu item which calls a high-level BUSINESS PROCESS. This high-level BUSINESS PROCESS in turn *refines into* other RULE SETS which may in their own turn *use* other RULE SETS or 3GL MODULES. A RULE SET may *access* a DATA VIEW through which it can *communicate* with a SCREEN DEFINITION, or *create* a REPORT. The DATA VIEW, in turn, will be *defined* by one or more DATA ELEMENTS. A RULE SET or 3GL MODULE may also *access* a FILE.[5]

These relationships, like the objects themselves, reside in the object repository. Every such relationship is represented by a DB2 database entry, and collectively, this database of relationships constitutes the application *meta-model* -- the abstract structural map of the application system, as shown in Figure 2.

---

[4]Veryard has noted that considerable effort must still be expended to make code reuse work effectively. "[Reusable] code may be more difficult to design and test, and there is always a temptation for the designer to develop something new, rather than take the trouble to investigate and implement something that already exists" ([68], p. 229).

[5]The verbs in the HPS 4GL language that we have already mentioned include *use, own, communicate, create, include* and *access*. The reader now should have a feel for how the nouns and verbs go together, without focusing on details of the syntax that HPS enforces.

INSERT FIGURE 2 ABOUT HERE

We can use this general meta-model 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 is 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 (software reuse, complexity, function points, etc.) is already contained in usable form in the meta-model. This information would have to be deduced from a detailed (and probably manual) analysis of the source code developed in a third generation environment.

## 3. FPA: A FUNCTION POINT ANALYZER FOR ICE

ICE satisfies two important prerequisites for the automation of function point analysis. *First*, the object repository, and its application meta-models, allow us to automate the identification of all software belonging to a given system. 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 software by multiple applications.

*Second*, the design of ICE's object-based 4GL 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.

### 3.1. Mapping Function Point Concepts to ICE Objects

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*. These include External Inputs, External Outputs, External Interfaces and Queries. Logical Internal Files constitute the fifth function type; they measure data stores internal to the application. ICE decomposes object and entity-relationship definitions into specific functional roles, and there is a well-defined mapping from ICE objects or relationships to function counts. This is illustrated in Figure 3, which also provides a conceptual representation of what we mean by the "application boundary."

---

INSERT FIGURE 3 ABOUT HERE

---

#### 3.1.1. External Inputs

A SCREEN with an output DATA VIEW (i.e., a SCREEN which sends data back to the invoking RULE SET) is an External Input. A FILE access is an input if the FILE is external to the system. The complexity of the External Input is determined by examining the number of DATA VIEWS and ELEMENTS or, in the case of a FILE access, the number of keys instead of DATA VIEWS.

### 3.1.2. External Output

A SCREEN with an input DATA VIEW (i.e., a SCREEN which receives data from the RULE SET which calls it) is an External Output, as is a REPORT or an output to an external FILE. Again, the complexity of the External Output is determined by examining the number of DATA VIEWS and ELEMENTS or, in the case of a FILE access, the number of keys instead of DATA VIEWS.

### 3.1.3. Queries

A SCREEN which allows a user to access data, but not to update it (this can be determined by comparing the FIELDS used in its input and output VIEWS) represents a Query. (Queries have lower function counts than the input/output combination of update-capable screens.) The complexity of a query is determined by examining the number of DATA VIEWS and ELEMENTS.

### 3.1.4. Logical Internal Files

A Logical Internal File is defined in the following manner: A FILE is internal to an application if some RULE SETS and 3GL MODULES that access the FILE are also internal to the application. (FPA checks which RULE SETS or 3GL MODULES access the FILE and examines if they are subordinate to the high-level RULE SET or BUSINESS PROCESS that defines the application). The complexity of a Logical Internal File is determined by the number of keys and DATA ELEMENTS it is defined to possess.

FPA also counts DATA DOMAINS, a special case of FILES with ICE. DATA DOMAINS are used by an application to validate or verify the values a user inputs and are analogous to sets.

### 3.1.5. External Interfaces

A FILE that is accessed by a RULE SET or a 3GL MODULE which is not part of the application represents an External Interface, as well as either an External Input or an External Output. The complexity of the interface is determined by the number of DATA ELEMENTS and keys.

Each function type gives rise to a number of function counts which depend upon its type and complexity. The function count of a system is the sum of the function counts of its component function types. See Table 1 below.

---

INSERT TABLE 1 ABOUT HERE

---

In most third-generation languages, a single program may easily give rise to any or all of the five function types, possibly multiple times. The only way to determine the functionality which it represents is to read and understand it. Each ICE object, by contrast, fills a limited role. That role, as we have seen, may be determined by an examination of the meta-model and of the data definitions associated with the object.

### 3.2. Computing Function Points in FPA

The *Function Point Analyzer* (**FPA**) has three main components that execute the function point analysis methodology: an Object Identifier, a Function Counter and a Complexity Factor Counter. These components are shown in Figure 4.

INSERT FIGURE 4 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 FUNCTION, PROCESS or high-level RULE SET chosen by the project manager that defines 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 their relationships, to function types and complexities, 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.*

FPA determines the fourteen complexity factors from function point analysis 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 of FPA, the objective measures are computed in parallel with managers' inputs, which only take a few minutes. When they have been sufficiently validated through use of FPA, the corresponding manual inputs will be replaced entirely, where possible. Each complexity factor has a separate input response screen that displays a definition of the complexity factor. See Figure 5.

INSERT FIGURE 5 ABOUT HERE

This can help a project manager who may not be familiar with function point analysis to give accurate and consistent responses.

The sequence of computation, then, is:

(1) *The Object Identifier traverses the meta-model in order to identify the objects and relations which may represent functionality.*

(2) *The Function Counter computes and sums the function count scores associated with those objects and relations.*

(3) *The Complexity Factor Counter computes the environmental complexity of the application on the basis of user inputs, and generates an adjustment factor for the function count. The maximum adjustment, positive or negative, is 35%.*

(4) *Function points are computed as the product of function counts and the environmental complexity adjustment factor (Refer to Appendix 1.)*

Thus, an automated function point analysis for a given application system would result in the collection of all data needed to compute function counts and make the environmental complexity adjustment. The output can be stored to an historical database for future use by project, department and senior IS managers. (An illustration of how FPA works in the context of the Broker Sales Reporting System is

presented in Sidebar 1, Figures 5 and 6, and Tables 1 to 4 at the end of this paper.)

---

INSERT SIDEBAR 1, FIGURES 5 AND 6, AND TABLES 1 TO 4 ABOUT HERE

---

## 4. SRA: A SOFTWARE REUSE ANALYZER FOR ICE

Software reuse is known to be a major source of productivity gains and cost reduction in software development operations [3, 43, 49, 60]. A study conducted at the Missile Systems Division of the Raytheon Company found that over 60% of procedural code was repeated in multiple applications [9], and reuse levels in non-manufacturing and non-engineering business applications (where less technical specificity is required) may even be greater. Considering the high costs of software development pervasively reported in the popular press, reuse represents a source of savings that managers are increasingly interested in tapping.

Due to the difficulties associated with identifying reuse in 3GL and 4GL environments, efforts to implement and manage successful reuse programs have been stymied in many organizations [31, 41]. Although certain types of explicit reuse (e.g., reuse of data definition files) have been easy to identify, most reuse in these environments is buried within programs where it is not easily identified without considerable manual effort.

An integrated, object-based CASE environment provides two major aids to the implementation and measurement of reuse. *First*, the code exists at a level of granularity more conducive to the implementation of software reuse. While it is rare that an entire 3GL program will prove reusable, such programs frequently contain routines which *could* be reused with 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. *Second*, the integrated environment serves to support the control and the measurement of software reuse. With the design of the entire system stored centrally along with the software itself, an instance of reuse becomes readily identifiable: it is simply the repeated invocation of an object within the repository.

To provide managers with information on software reuse, we designed and developed a facility within ICE called the *Software Reuse Analyzer* (**SRA**). SRA analyzes an existing software application, reporting the levels of reuse for the various elements comprising the application. Like FPA, SRA identifies all the relevant objects for a given analysis by systematically navigating the hierarchy of calling relationships within the repository.

### 4.1. Measurement of Software Reuse

According to an appendix to the IEEE Computer Society's recent draft proposal for a "Standard for Software Productivity Metrics", reused software may be measured by the number of *logical source statements* (LSS) or *physical source statements* (PSS) incorporated or ported unmodified into an application system [61]. New software, then, may be measured by the number of LSS or PSS that were created or modified for the application system.[6] We have adapted this taxonomy for ICE: A pre-existing object is considered to be reused if it is incorporated unmodified into an application system that is designed in accordance with another application system.[7] In ICE terms, such reuse is implemented simply by adding a

---

[6]The appendix is not formally a part of the IEEE standard.

[7]Parnas conceptualized the manner in which an operating system or a program carries out its processes by distinguishing between two primary operations upon modules, "invokes" and "uses". "Uses" requires the actual execution of a software object in order for the operation to conclude; "invokes" is meant to indicate a conditional call to a software object. Parnas further argues that it is possible to formally specify the operation of a software application in terms of a module hierarchy that is loop-free, while maintaining a program structure (more formally called a "uses hierarchy") that encourages software reuse and avoids the trap of highly interdependent system parts [53, 54]. In HPS, reuse is the inclusion of a previously defined object within an application system's "uses hierarchy". The reader who wants to obtain additional familiarity with the principles

new relationship to the meta-model, thus calling a previously written object. Once all the objects within an application have been identified, SRA computes a number of managerially useful reuse metrics which are based upon counts of new objects and reused objects in an application system.

A number of studies have observed that the potential for reuse in software development extends far beyond the reuse of source lines of code. For example, Jones [32] suggested the following kinds of reuse in software development operations: data, architecture, designs, programs and common subsystems and modules. Kernighan [35] examined the same issues in the context of the UNIX operating system and identified potential reuse at the code library, programming language, program and system levels. Bollinger and Pfleeger [10] add documentation, test data, and intangibles such as specialized learning to this list. The focus of this paper is limited to reuse of objects, although ICE stores information about the functional and technical design of a system as well[8].

As Hall [28] has pointed out, metrics based on counts of instances of reuse may be deficient in addressing many of the managerial questions concerning reuse:

> *[The] developer needs to ascertain what sort of reuse is meant. Is it the number of times the code is incorporated into other code? The number of times the code is executed? A combination, the number of times the incorporating code is executed? A figure of merit reflecting the value or utility or saving rather than being a simple count of uses?* [28, p. 41]

---

of system decomposition should refer to [51] and [52]. For a broader treatment of the issues of reusability and reuse see the surveys by Tracz [67], Hooper and Chester [30], Frakes et al. [24] and Norman et al. [48]. The last two were presented as panel discussions at the *13th International Conference on Software Engineering* in May 1991.)

[8]One of the major benefits of object-oriented design is that the reuse of an object can imply the reuse of elements of the system's design as well as its coding, to a far greater degree than is generally true for procedure-oriented design [11].

In the process of designing SRA, we identified three primary types of issues that its software reuse metrics would need to address:

* What objects are being reused?

* How effective is a particular system or environment in promoting software reuse?

* What is the impact of this reuse on productivity and development costs?[9]

As a result, we present metrics to address all three kinds of questions: *reuse leverage metrics*, *reuse classification metrics*, and *reuse value metrics*, respectively.

#### 4.1.1. Leverage Metrics

*New Object Percent* measures the leverage achieved through reuse. It is the proportion of the objects within a system that actually had to be written for the system. (The rest of the objects represent instances of reuse, and hence cost savings attributable to reuse.) We define New Object Percent within an application as:

$$\textit{NEW OBJECT PERCENT} = \frac{\textit{NUMBER OF NEW OBJECTS BUILT}}{\textit{TOTAL NUMBER OF OBJECTS USED}} * 100\%$$

To illustrate this metric, let us consider a system consisting of 400 objects, of which 100 had to be programmed from scratch. The New Object Percent is 100/400 * 100% = 25%, meaning that for every

[9]For discussions of the use and value of economics-based approaches to the evaluation of software development performance, see Banker and Kauffman [4], Boehm [8], Kang and Levy [33], and Levy [39]. Gaffney and Durek's analysis of the cost impact of reusable software [25, 26] also suggests a strong rationale for creating such metrics.

four objects within the system, only one had to actually be built for that system.[10] Knowing the extent to which new software must be developed across a firm's applications provides management with the opportunity to attempt to mandate what levels are desirable and manage software development activities to achieve them.

We may say that the New Object Percentage is 25% or, equivalently, that the average object is used four times.[11] We refer to this metric as *Reuse Leverage*, which we formally define as:

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

These measures of reuse can be applied at several levels of analysis. In computing separate reuse leverage factors for different object types, for example, we might find that the summary reuse factor of 25% aggregates a reuse leverage factor of 40% for RULE SETS and 15% for SCREEN DEFINITIONS. Since RULE SETS take far more time than SCREEN DEFINITIONS to write, the aggregate measure in this example underestimates the benefits of reuse.

#### 4.1.2. Classification Metrics

For most purposes, we include in our computation of software reuse, any object which is found in the repository, rather than rewritten from scratch. For some managerial purposes, however, we will wish to distinguish *internal reuse* from *external reuse*. Reuse is internal if an object created for a system is used

---

[10]Note that we have diverged from our initial definition in that a pre-existing object which is invoked without modification is considered to constitute an instance of reuse whether it originated in a different system, or whether it was just written for the current system and then used more than once. This distinction will be dealt with in the next section.

[11]Although this metric has less desirable analytical qualities, our experience has been that managers often find it easier to understand.

multiple times within the system. It is external if an object from a different system is used one or more times within the new system. ICE considers an object to be owned by the system for which it was originally created, and the software reuse analyzer has access to that information. (Almost all the reuse displayed in Figures 7 and 8 is internal.) While both kinds of reuse are of equal value (strictly speaking, external reuse guarantees the developer that the object has been tested elsewhere prior to being made more widely available in the repository), different managerial policies may be required to encourage them.

---

INSERT FIGURES 7 AND 8 ABOUT HERE

---

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, may depend more upon the quality of the indexing system used to help programmers to identify existing objects which they might be able to reuse [6, 21, 55]. When reuse metrics are being computed for all the objects within the repository, all reuse is internal, by definition.

*Reuse classification metrics* allow us to assess and compare system reuse by classifying a system's objects by source. Some examples are shown below:

$$\textit{EXTERNAL REUSE PCT} = \frac{\textit{NUMBER OF OBJECTS OWNED BY OTHER SYSTEMS}}{\textit{TOTAL NUMBER OF OBJECTS USED}}$$

$$\textit{INTERNAL REUSE PCT} = 100\% - \textit{NEW OBJECT PCT} - \textit{EXTERNAL REUSE PCT}$$

Internal reuse percentage, here, is interpreted as the proportion of occurrences of objects written for an application (not counting the first occurrence of each object) compared to the total number of objects used in the application. These metrics can be modified as in the preceding section to reflect differences in the

relative costs of developing the objects.

#### 4.1.3. Value Metrics

We also wish to measure the cost implications of reuse. The other metrics we have discussed value all instances of reuse equally, and do not consider the fact that some objects may represent considerably greater costs, or considerably more functionality, than others. *Reuse value* can be determined using two general approaches.

*The Object Reuse Standard Cost Method* computes a reuse value by estimating the cost saving attributable to reuse. A standard cost is assigned to each object type -- based on actual site experience -- and the number of reuse instances for each object type is multiplied by the appropriate standard cost. (In practice, different standard costs can be estimated for objects of low, average, and high complexity.) The computation can be accomplished as a by-product of the reuse leverage and reuse classification analyses, and requires no additional automation, other than reference to a table of standard costs, which may differ from firm to firm.

This method can be applied to a single application or to the entire repository. Analysis yields the proportion of the total software costs that have been avoided through reuse, calculated as:

$$REUSE\ VALUE = 1 - \frac{\sum_{j=1}^{J*} OBJECT\text{-}STD\text{-}COST_j}{\sum_{j=1}^{J} OBJECT\text{-}STD\text{-}COST_j}$$

where

$OBJECT\text{-}STD\text{-}COST_j$ = *standard (average cost) in person days for building object type j;*

$J$ = *total number of occurrences of objects in an application meta-model hierarchy;*

$J^*$ = *total number of unique objects built for this application.*

This metric differs from the similar one proposed by Gaffney and Durek [26], in that it does not consider reuse costs. In the ICE environment, these costs are typically very low: If an object can be reused without modification, the cost of including it in an application is negligible.[12] This, of course, is not always the case; sometimes there is a near match, resulting in the reuse of the existing object to template a new one. The value of this "hidden reuse" is not included in this metric.

*The Function Point Reuse Standard Cost Method* measures the proportion of the application's function points which are attributable to reuse. A value can be derived from this figure by applying a single standard cost per function point. This approach is primarily of interest at higher managerial levels than that of the project manager.

In ICE, as was seen in our discussion of the Function Point Analyzer, although development effort may depend upon the number and complexity of the objects in the repository, the functionality of the system (as measured by function points) depends upon the relationships in the meta-model. Every time we add a new call to an object which is already in use, we are adding a computable number of function points to the system, without writing any new objects. We can represent the value of function point reuse by determining the total costs associated with building all of the function points in an application (either from real project costs or from organization-wide standard costs for building a function point) and then

---

[12] There is still a search cost associated with reuse. Programmers must identify appropriate objects, and then spend enough time studying them to confirm that they are appropriate for reuse. The reader should refer to Dunn and Knight [19] and Fischer, Henninger and Redmiles [22] for useful, current perspectives on the problem of searching for reusable software. For additional background on the MITRE Corporation and the Software Productivity Consortium's research program on the economics of software reuse, see [15, 16, 25].

determining the proportion that results from reuse. The associated reuse value metric is shown below:

$$REUSE\ VALUE = 1 - \frac{\sum_{j=1}^{J^*} FP_j}{\sum_{j=1}^{J} FP_j}$$

where

$FP_j$ = *the number of function points associated with relation j;*

$J$ = *the total number of relations in an application meta-model hierarchy;*

$J^*$ = *the total number of unique objects (and hence, the total number of relations which are first-time calls to those objects, rather than instances of reuse) built for this application.*

Since function points are the basis for ICE productivity measurement, this reuse value metric gives us a measure of the proportion of system functionality, and hence of developers' output and productivity, which is attributable to reuse. Unlike the object reuse standard cost value metric, it has not yet been implemented in SRA.

### 4.2. SRA Architecture

The operations of the Software Reuse Analyzer parallel those of the Function Point Analyzer. *First*, SRA

*identifies* the objects used by a given application the same way that the Function Point Analyzer does. The repository contains a complete meta-model describing the relationships between application objects, and SRA uses it to trace all the objects which are called, directly or indirectly, by the application under analysis. As with FPA, the scope of the analysis is determined by the user at the time of execution. It can include the entire contents of the repository, a small or large set of application systems, or even a subset of a single system. The ability to start anywhere in the hierarchy provides SRA with a great deal of power for addressing managerial concerns about reuse. For example, reuse may be analyzed for a specific type of application, for a given project team, for a given manager, etc. It also facilitates research into what factors contribute to increased reuse.

*Second*, once the set of objects has been identified, SRA *classifies* the objects. The repository contains information to not only identify the objects called by a given object, but also to identify the source of each object. If a given object was originally written for a different system (i.e., one beyond the scope of the current analysis) then it is an instance of external reuse. If it was written for the system being analyzed, then the first time it is encountered by the analyzer it is classified as newly-written software, while subsequent encounters are classified as instances of internal reuse. *Finally*, SRA *computes* multiple reuse metrics for management. (An illustration of how SRA computes the software reuse metrics in the context of the Broker Sales Reporting System is presented in Sidebar 2, Figures 7 and 8, and Tables 5 and 6 at the end of this paper.)

---

INSERT SIDEBAR 2, FIGURES 5 AND 6, AND TABLES 5 AND 6 ABOUT HERE

---

## 5. CONCLUSION

We have described two automated software analyzers: a Function Point Analyzer (**FPA**) and a Software Reuse Analyzer (**SRA**). In the process of thinking through the conceptual design problems and testing the analyzers, we were able to come to an improved understanding of the nature of the productivity gains attributable to CASE tools. Such productivity gains are typically thought of as the result of being able to produce the desired software more quickly and cheaply. In fact, our analysis reveals that much of the gain is represented by the production of *functionality* which, without the improved tools, might well not exist.

The Integrated CASE Environment (**ICE**) automatically provides many capabilities that would require considerable programmer resources in a traditional programming environment, such as the automation of inter-platform communications, the automatic generation of "HELP" messages for every field on a screen, and the automatic translation of any table to graphical format (an especially useful capability for traders who use on-line, real-time trader workstations in investment banking firms).

In many cases, designers in a 3GL environment would probably choose to do without these capabilities, rather than expend the cost and effort needed to implement them without the appropriate CASE support. Thus, the comparisons which are frequently cited between the cost of producing a system using a given CASE technology and the cost which traditionally would have been incurred may be misleading in the productivity advantage they appear to indicate for the CASE tools. At the same time, they may tend to overlook the superior functionality and user-friendliness which may be expected to accompany CASE development.[13]

---

[13]This raises a related issue. The function types which are assigned the highest weights in function point analysis are those which are most difficult to implement in a 3GL. But often these are not difficult at all, with CASE support. Function points may be useful, then, in answering the question "What would this system have cost to develop without CASE?" But a recalibrated measure may be required in order to estimate costs within a given CASE environment. See Banker, Kauffman and Kumar [5] for a discussion of a new approach called *object point analysis* that addresses this issue for an object-based CASE

### 5.1. Contributions

This paper had multiple objectives. We wished to report on our automation of function point and software reuse metrics -- automation which has not been possible in traditional programming environments. We wished to generalize from our experience, to identify the features of the CASE environment that make this automation possible. We wished to report on the implications that this research has for our understanding of software productivity in an integrated CASE environment.

The Function Point Analyzer and the Software Reuse Analyzer described in this paper represent efforts that are on the frontier of the design and development of automated software metrics facilities in an integrated CASE tool environment. Their implementation was made possible by two key features of the repository object-based integrated CASE environment. The first of these features is the repository itself, which contains not only all the software and data used by the applications, but also an indexing system (in this case, the meta-model) which allows us to identify the software and files belonging to each application, as well as the key relationships between them which result in application functionality. It is conceptually possible for this information to be maintained (within a repository or otherwise) by a non-integrated CASE tool, but we consider it improbable that the integrity of the information could or would be maintained in such circumstances. The second feature is the repository object-based CASE environment and its 4GL. The organization of the software into objects of limited and clearly defined functionality has enabled us to compute function points and to identify reuse without having to actually analyze and understand the code itself.

We discussed three classes of metrics for assessing software reuse: leverage metrics, value metrics and classification metrics. The first two of these metrics match the efficiency and effectiveness dimensions of standard performance evaluation approaches. These measures help managers to distinguish between

---

environment.

*aggregate levels of reuse* that are achieved in projects or by areas of the firm's software development operations, as well as *reuse of individual objects* that are especially costly to build. Moreover, we have suggested that a variety of metrics that triangulate on the key management problems are of interest here: a unitary measure of software reuse lacks the power to answer the questions that we found to be important to managers.

We also showed how traversing a hierarchical meta-model of a repository object-based system enables us to identify objects used by a given system or subsystem, and define reuse which is internal to the hierarchy (for example, software reused within a program or an application) or which is external to it. Initial analysis that we have conducted at our research site suggests that this classification is important to managers wishing to encourage software reuse. It appears that internal reuse will proliferate where the technology supports it: programmers routinely reuse software from one part of an application in another. Software that is external to the system, however, tends to be written by other programmers, and different technical support and organizational incentives are needed in order to motivate programmers to seek out external reuse opportunities [6].

Clearly, these questions are only the starting point for a rich, new management agenda to better understand and control CASE-based development [67]. Yet, we are already left with some answers we did not have before we began this research. We have learned that the data collection and analysis needed in order to control software costs can be automated. We have identified features of CASE systems which support such automation. And we have begun to understand the issues involved in measuring output and reuse in such environments [7].

### 5.2. Future Research on Productivity and Software Metrics

Our research raises questions about the continued usefulness of function points -- a measure designed and

calibrated for use in traditional 3GL environments. Are they still useful as predictors of programming costs within an integrated CASE environment? Are they useful as a means of exercising managerial control in such an environment? Can they be used to predict staffing requirements or future maintenance requirements? Could they be made more useful by recalibrating and fine-tuning them for new conditions?

In a similar vein, our development of the Software Reuse Analyzer gave us an improved understanding of software reuse. Our tests of SRA confirmed that commercial application systems built using ICASE offer tremendous scope for software reuse. If the average object is used five times, this can mean an 80% reduction in the cost of programming and unit testing, and we have observed such reuse levels for some systems built using ICE. However, initial analysis suggests that, even here, only a fraction of the potential for reuse is being tapped. Programmers tend to only reuse software with which they are personally familiar, so that relatively low levels of external reuse are observed.

We are now in the process of formulating research to deal with the questions raised by these observations -- questions that have been examined elsewhere, for example, in the context of the Department of Defense's Joint Integrated Avionics Working Group (**JIAWG**) on software reuse [58], the U.S. Army Information Systems Software Development Center's RAPID Center Library (**RCL**) software reuse library for Ada [69], Magnavox's U.S. Army Advanced Field Artillery Tactical Data System (**AFATDS**) project [13] and GTE Data Services' software asset management program [62], and other efforts reviewed by Hooper and Chester [30] and Tracz [66]. These include the following: How can software reuse be supported, encouraged and motivated? What aspects of the software are conducive to reuse and most likely to pay off in the long-term? What programming and managerial practices provide the proper incentives for software reuse?

The automated report generation capabilities of the FPA and SRA enable us to pursue research questions that were simply beyond the scope of prior research in terms of cost and availability of data. What can we

learn about software development productivity in this environment? Do productivity gains change with CASE or application-specific experience? With the passage of time and the accretion of maintenance changes? What are the features of CASE tools that best encourage productivity? Which slow it down?

The questions raised here are the basic questions that software development managers will have to answer: What works? What doesn't work? How well does a given software solution work? How can it be made to work better? The availability of appropriate metrics makes it possible for managers to start answering these questions.

## REFERENCES


1. A. J. Albrecht, "Measuring Application Development Productivity," in *Proc. Joint SHARE, GUIDE, and IBM Application Development Symp.*, IBM, pp. 83-92, October 1979.

2. A. J. Albrecht and J. E. Gaffney, "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," *IEEE Trans. Software Eng.*, vol. 9, no. 6, pp. 639-647, November 1983.

3. U. Apte, C. S. Sankar, M. Thakur, and J. Turner, "Reusability Strategy for Development of Information Systems: Implementation Experience of a Bank," *MIS Quarterly*, vol. 14, no. 4, pp. 421-431, December 1990.

4. R. D. Banker, and R. J. Kauffman, "Reuse and Productivity: An Empirical Study of Integrated Computer Aided Software Engineering at The First Boston Corporation," *MIS Quarterly*, vol. 15, no. 3, pp. 375-401, September 1991.

5. R. D. Banker, R. J. Kauffman, and R. Kumar, "An Empirical Assessment of Object-Based Output Measurement Metrics in Computer Aided Software Engineering," *Jrnl. Mgmt. Info. Sys.*, vol. 6, no. 3, Winter 1991-92.

6. R. D. Banker, R. J. Kauffman, and Zweig, D., "Factors Affecting Code Reuse: Implications for a Model of Computer Aided Software Engineering Development Performance," Ctr. for Research on Info. Sys., Stern School of Business, New York Univ., December 1990.

7. R. D. Banker, R. J. Kauffman, and Zweig, D., "Repository Evaluation of Software Reuse," *IEEE Trans. Software Eng.*, forthcoming.

8. B. W. Boehm, *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

9. B. W. Boehm, and P. N. Papaccio, "Understanding and Controlling Software Costs," *IEEE Trans. Software Eng.*, vol. 13, no. 10, pp. 1462-1477, October 1988.

10. T. B. Bollinger, and S. L. Fleeger, "Economics of Reuse: Issues and Alternatives," *Info. Software Tech.*, vol. 32, no. 10, pp. 643-652, December 1990.

11. G. Booch, "What Is and What Isn't Object-Oriented Design," *Ed Yourdon's Software Jrnl.,* vol. 2, nos. 7-8, pp. 14-21, Summer 1989.

12. B. M. Bouldin, "CASE: Measuring Productivity -- What Are You Measuring? Why Are You Measuring It?" *Software Mag.,* vol. 9, no. 10, pp. 30-39, August 1989.

13. H. B. Carstensen, Jr., "A Real Example of Reusing Ada Software," in *Proc. Conf. on Software Reusability and Maintainability,* Tysons Corner, VA: The National Institute for Software Quality and Productivity, Inc., 1987.

14. P. S. Chen, "The Entity-Relationship Approach to Information Modeling and Analysis," in *Proc. 2nd Intl. Conf. on the Entity-Relationship Approach*, Saugus, CA: ER Institute, held in Washington, DC, October 12-14, 1981.

15. R. D. Cruickshank and J. D. Gaffney, Jr., "An Economics Model of Software Reuse," presented at *MITRE-Washington Econ. Analysis Ctr. Conf. on Analytical Methods in Software Eng. Econ. I*, Washington, DC, July 1991.

16. R. D. Cruickshank, and J. D. Gaffney, Jr., "A Software Cost Model of Reuse within a Single System," presented at *MITRE-Washington Econ. Analysis Ctr. Conf. on Analytical Methods in Software Eng. Econ. II*, Washington, DC, July 1992.

17. G. B. Davis, "Commentary on Information Systems: Productivity Gains from Computer Aided Software Engineering," *Acct. Horizons*, vol. 2, no. 2, pp. 90-93, June 1988.

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

19. M. F. Dunn and J. C. Knight, "Software Reuse in an Industrial Setting: A Case Study," in *Proc. 13th Intl. Conf. Software Eng.*, Austin, TX: IEEE Comput. Soc. Press, pp. 329-338, May 13-17, 1991.

20. E. M. Dusink, "Towards a Design Philosophy for Reuse," in *Proc. Reuse in Practice Workshop*, J. Baldo and C. Braun (eds.), Pittsburgh, PA: Software Engineering Institute, 1989.

21. G. Fischer, "Cognitive View of Reuse and Re-design," *IEEE Software*, vol. 4, no. 4, pp. 60-72, July 1987.

22. G. Fischer, S. Henninger and D. Redmiles, "Cognitive Tools for Locating and Comprehending Software Objects for Reuse," in *Proc. 13th Intl. Conf. Software Eng.*, Austin, TX: IEEE Comput. Soc. Press, pp. 318-328, May 13-17, 1991.

23. J. T. Fisher, "IBM's Repository: Can Big Blue Establish OS/2 EE As the Professional Programmer's Front End?" *DBMS*, pp. 42-49, January 1990.

24. J. B. Frakes, T. J. Biggerstaff, R. Prieto-Diaz, K. Matsumura, and W. Shaefer, "Software Reuse: Is It Delivering?" in *Proc. 13th Intl. Conf. Software Eng.*, Austin, TX: IEEE Comput. Soc. Press, pp. 52-59, May 13-17, 1991.

25. J. E. Gaffney, Jr., "An Economics Foundation for Software Reuse," Herndon, VA: Software Productivity Consortium, 1989.

26. J. E. Gaffney, Jr., and T. A. Durek, "Software Reuse -- Key to Enhanced Productivity: Some Quantitative Models," *Info. Software Tech.,* vol. 31, no. 5, pp. 258-267, June 1989.

27. G. W. Grammas and J. R. Klein, "Software Productivity As a Strategic Variable," *Interfaces*, vol. 15, no. 3, pp. 116-126, May-June 1985.

28. P. A. V. Hall, "Software Components and Reuse -- Getting More Out of Your Code," *Info. Software Tech.*, vol. 29, no. 1, pp. 38-43, January-February 1987.

29. A. Hazzah, "Making Ends Meet: Repository Manager," *Software Magazine*, pp. 59-72, December 1989.

30. J. W. Hooper, and R. O. Chester, *Software Reuse: Guidelines and Methods.* New York, NY: Plenum Press, 1991.

31. E. Horowitz, and J. Munson, "An Expansive View of Reusable Software," *IEEE Trans. Soft. Eng.,* vol. SE-10, no. 5, pp. 477-487, September 1985.

32. T. C. Jones, "Reusability in Programming: A Survey of the State of the Art," *IEEE Trans. Software Eng.*, vol. SE-10, no. 5, pp. 484-494, September 1984.

33. K. C. Kang and L. S. Levy, "Software Methodology in the Harsh Light of Economics," *Info. Software Tech.*, vol. 31, no. 5, pp. 239-249, June 1989.

34. C. F. Kemerer, "Reliability of Function Points Measurement: A Field Experiment," working paper, Sloan School of Management, MIT, December 1990.

35. B. W. Kernighan, "The UNIX System and Software Reusability," *IEEE Trans. Software Eng.*, vol. SE-10, no. 5, pp. 513-518, September 1984.

36. B. J. Kitaoka, "Managing Large Repositories for Reuse," in *Proc. Reuse in Practice Workshop*, J. Baldo and C. Braun (eds.), Pittsburgh, PA: Software Engineering Institute, 1989.

37. R. G. Lanergan and C. A. Grasso, "Software Engineering with Reusable Designs and Code," *IEEE Trans. Software Eng.*, vol. SE-10, no. 5, pp. 498-501, September 1984.

38. R. G. Lanergan and B. A. Poynton, "Reusable Code: The Application Development of the Future," in *Proc. IBM SHARE/GUIDE Software Symp.*, Monterey, CA: IBM, October 1979.

39. Levy, L. S., *Taming the Tiger: Software Engineering and Software Economics*. New York: Springer Verlag, 1987.

40. G. C. Low and D. R. Jeffrey, "Function Points in the Estimation and Evaluation of the Software Process," *IEEE Trans. Software Eng.*, vol. 16, no. 1 , pp. 64-71, January 1, 1990.

41. R. F. Mathis, "The Last 10 Percent." *IEEE Trans. Software Eng.*, vol. SE-12, no. 6, pp. 705-712, June 1986.

42. Y. Matsumoto, "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels," *IEEE Trans. Software Eng.*, vol. SE-10, no. 5, pp. 502-512, September 1984.

43. C. McClure, *The Three Rs of Software Automation: Re-engineering, Repository, Reusability*. Englewood Cliffs, NJ: Prentice Hall, 1992.

44. B. Meyer, "Reuse: The Case for Object-Oriented Design," *IEEE Software*, vol. 4, no. 2, pp. 50-64, March 1987.

45. B. Meyer, *Object Oriented Software Construction.* New York: Prentice Hall, 1988.

46. J. M. Neighbors, "The DRACO Approach to Constructing Software from Reusable Components," *IEEE Trans. Software Eng.,* vol. SE-10, no. 5, pp. 564-574, September 1984.

47. R. J. Norman and J. F. Nunamaker, Jr., "CASE Productivity Perceptions of Software Engineering Professionals," *Commun. Ass. Comput. Mach.,* vol. 32, no. 9, pp. 1102-1108, September 1989.

48. R, J. Norman, W. Stevens, E. J. Chikofsky, J. Jenkins and B. L. Rubenstein, "CASE at the Start of the 1990s," in *Proc. 13th Intl. Conf. Software Eng.*, Austin, TX: IEEE Comput. Soc. Press, pp. 128-142, May 13-17, 1991.

49. J. F. Nunamaker, Jr., and M. Chen, "Software Productivity: A Framework of Study and an Approach to Reusable Components," in *Proc. 22nd Hawaii Intl. Conf. Sys. Sci.*, Hawaii, IEEE, pp. 959-968, January 1989.

50. J. F. Nunamaker, Jr., and M. Chen, "Software Productivity: Gaining Competitive Edges in an Information Society," in *Proc. 22nd Hawaii Intl. Conf. Sys. Sci.*, Hawaii, IEEE, pp. 957-958, January 1989.

51. D. L. Parnas, "A Technique for Software Module Specification with Examples," *Commun. Ass. Comput. Mach.*, vol. 15, no. 5, pp. 330-336, May 1972.

52. D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Commun. Ass. Comput. Mach.*, vol. 15, no. 12, pp. 1053-1058, December 1972.

53. D. L. Parnas, "Some Hypotheses About the Uses Hierarchy for Operating Systems," tech. rep., Technische Hochschule Darmstadt, Darmstadt, Germany, 1976.

54. D. L. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Trans. Soft. Eng.*, vol. SE-5, no. 2, pp. 128-137, March 1979.

55. R. Prieto-Diaz, "Classifying Software for Reusability," *IEEE Software*, vol. 4, no. 1, pp. 6-16, January 1987.

56. A. Pollack. "The Move to Modular Software," *New York Times*, pp. D1-2, Monday, April 23, 1990.

57. F. J. Polster, "Reuse of Software Through Generation of Partial Systems," *IEEE Trans. Software Eng.,* vol. SE-10, no. 5, pp. 402-416, September 1984.

58. D. J. Reifer, "Joint Integrated Avionics Working Group (JIAWG) Reusable Software Program Operational Concept Document," Torrance, CA: Reifer Consultants Inc., 1990.

59. J. A. Senn and J. L. Wynekoop, "Computer Aided Software Engineering in Perspective," Info. Tech. Mgmt. Ctr., Coll. Business Admin., Georgia State Univ., 1990.

60. V. Seppanen, "Reusability in Software Engineering," in *Tutorial: Software Reusability*, P. Freeman (ed.), Austin, TX: IEEE Comput. Soc. Press, pp. 286-297, 1987.

61. "Software Productivity Metrics Working Group of the Software Engineering Standards Subcommittee, Standard for Software Productivity Metrics," IEEE Computer Society, P1045/D5.0 (draft), March 8, 1992.

62. M. E. Swanson and S. K. Curry, "Implementing an Asset Management Program at GTE Data Services," *Info. Mgmt.*, vol. 16, 1989.

63. C. R. Symons, *Extended Function Points with Entity Type Complexity Rules.* London: Nolan, Norton and Co., 1984.

64. C. R. Symons, "Function Point Analysis: Difficulties and Improvements," *IEEE Trans. Software Eng.*, vol. 14, no. 1, pp. 2-10, January 1988.

65. W. Tracz, "Making Reuse a Reality," *IEEE Software,* vol. 4, no. 4, July 1987.

66. W. Tracz, "Ada Reusability Efforts: A Survey of the State of the Practice," in *Tutorial: Software Reuse -- Emerging Technology*, Austin, TX: IEEE Comput. Soc. Press, pp. 23-32, 1988.

67. W. Tracz, *Tutorial: Software Reuse -- Emerging Technology.* Austin, TX: IEEE Comput. Soc. Press, 1988.

68. R. Veryard, "Information and Software Economics," *Info. Software Tech.,* vol. 31, no. 5, pp. 226-230, June 1989.

69. T. Vogelsong, "Reusable Ada Packages for Information System Development (RAPID) -- An Operational Center for the Excellence of Software Reuse," in *Proc. Reuse in Practice Workshop*, J. Baldo and C. Braun (eds.), Pittsburgh, PA: Software Engineering Institute, 1989.

70. K. Zwanzig, *Handbook for Estimating Function Points.* GUIDE Project -- DP-1234, GUIDE International, November 1984.

## APPENDIX 1. THE FUNCTION POINT ANALYSIS PROCEDURE

### STEP 1: Identification of Function Types.

Identify each functionality unit and classify it into five user function types:

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

* *External Inputs* are data items sent by the user to the computer for processing, or to make additions, changes or deletions.

* *Queries* are simple outputs; they are direct inquiries into a database or master file that look for specific data, use simple keys, require immediate response, and perform no update functions.

* *Logical Internal Files* are data stored for an application, as logically viewed by the user.

* *External Interface Files* are data stored elsewhere by another application, but used by the one under evaluation.

This step yields a count for each of the five different function types.

### STEP 2: Classification of Simple, Average and Complex Function Types.

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

**APPENDIX 1. THE FUNCTION POINT ANALYSIS PROCEDURE (continued)**

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

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

$$\sum_{t=1}^{5}\sum_{c=1}^{3} \textit{FUNCTION-TYPE}_t * \textit{FUNCTION-COMPLEXITY-SCORE}_c.$$

**STEP 3: Adjusting FUNCTION-COUNTS by TECHNICAL-COMPLEXITY-FACTOR.**

The adjustment factor reflects application and environmental complexity, expressed as the degree of influence of fourteen characteristics (f) listed below. Each characteristic is rated on a scale of 0 to 5 (COMPLEXITY-FACTOR), and then all scores are summed. The *TECHNICAL-COMPLEXITY-FACTOR* (**TCF**) = .65 + (.01 * $\Sigma_{f=1 \text{ to } 14}$ COMPLEXITY-FACTOR$_f$). The fourteen factors are shown below.

| | | | |
|---|---|---|---|
| 1. | Data Communications | 8. | On-line Update |
| 2. | Distributed Functions | 9. | Complex Processing |
| 3. | Performance | 10. | Reuse |
| 4. | Heavily Used Configuration | 11. | Installation Ease |
| 5. | Transaction Rate | 12. | Operational Ease |
| 6. | On-line Data Entry | 13. | Multiple Sites |
| 7. | End User Efficiency | 14. | Facilities Changes |

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

## SIDEBAR 1. THE BROKER SALES REPORTING SYSTEM: INTRODUCTION

The Broker Sales Reporting System is a small (simplified) ICE application system that illustrates the concepts presented in this article. The system tracks and reports the sales activity of brokers in a small investment firm. The application has both online and batch capabilities designed to meet the needs of middle and senior management. Senior management is provided with summarized reports and inquiries. Middle management is provided with detailed reports and inquiries concerning the performance of individual brokers.

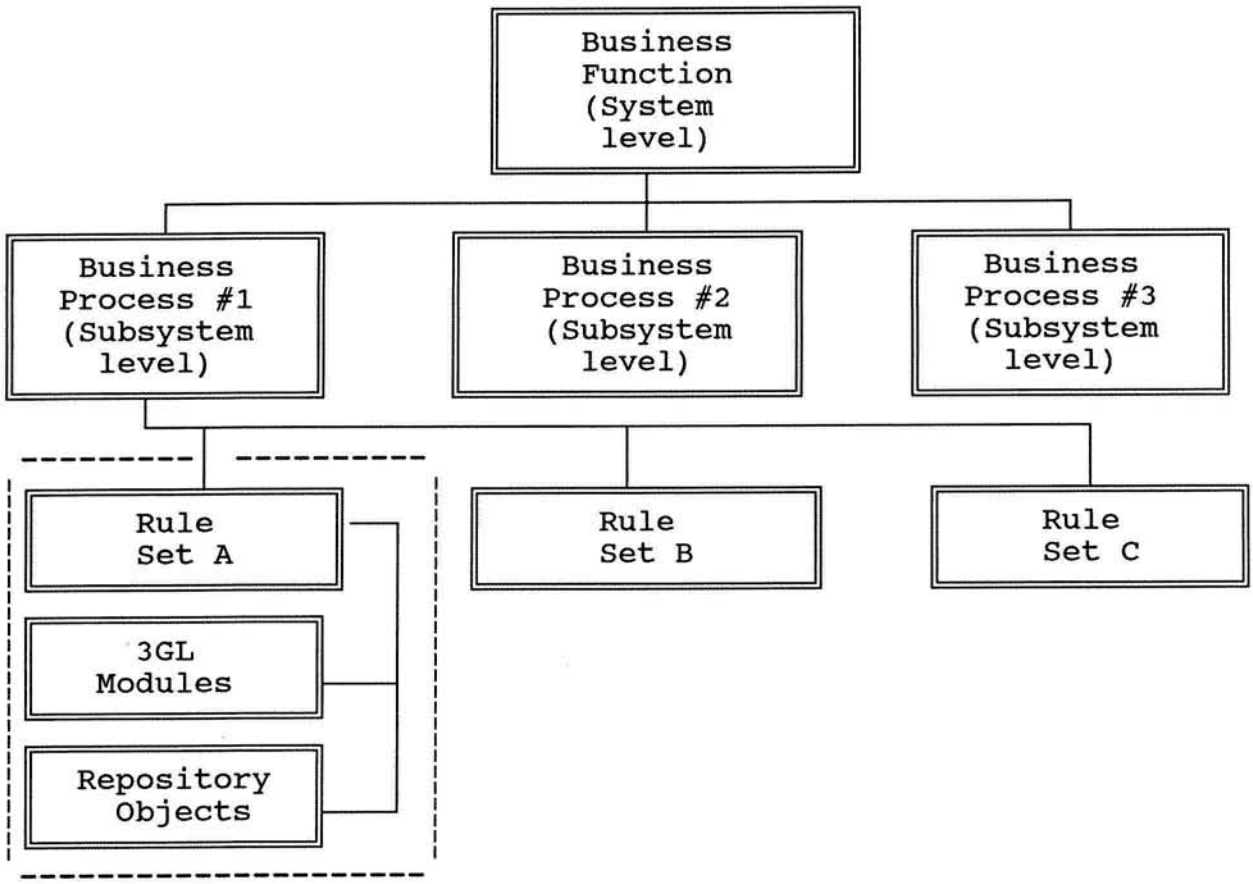
## SIDEBAR 2. THE SOFTWARE REUSE ANALYZER

The operation of the Software Reuse Analyzer will be illustrated for a subset of the Broker Sales Reporting System. By measuring software reuse one can measure the savings which may be realized by coding each object once and reusing it as necessary (Figure 7), instead of having to rewrite the code every time it is needed (Figure 8). A simple ratio of object counts yields the *leverage metrics*, NEW_OBJECT_PCT and REUSE_LEVERAGE. The REUSE_VALUE metric estimates the *savings* attributable to reuse, by considering not only the number of reused objects, but also the function points that they deliver. These can be equated with software development costs.

In principle, an integrated CASE system could be designed to capture actual costs for each object, as it is produced. This has not yet been implemented for ICE. Rather, a set of heuristics was developed, on the basis of interviews with software managers, for estimating the cost of an object (in days) based on its type and its complexity. The complexity is measured on a three-point scale (Simple, Average or Complex -- but not the same scale that is used for function point analysis) which is simple enough to automate. (These heuristics are in actual use by managers for project cost estimation; see Banker, Kauffman and Kumar [5] for a preliminary indication of their robustness.)

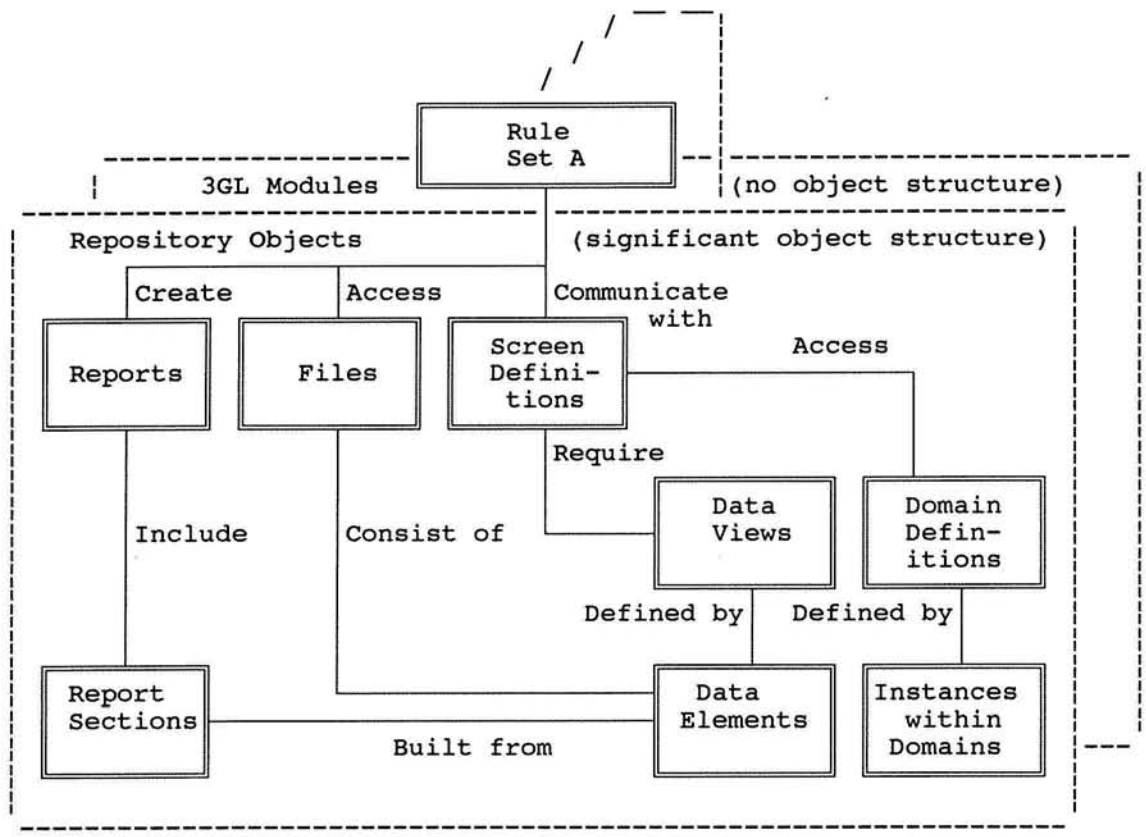
The Software Reuse Analyzer distinguishes between *internal reuse* -- the reuse of objects written for the current task -- and *external reuse* -- the reuse of objects previously written for different applications. We have observed relatively little reuse of code written by other programming teams, for other application systems. This suggests that special support may be required to encourage programmers to seek out opportunities for external reuse. Without that support, much of the potential software reuse goes unexploited.

**FIGURE 1. A REPOSITORY-BASED APPLICATION META-MODEL**



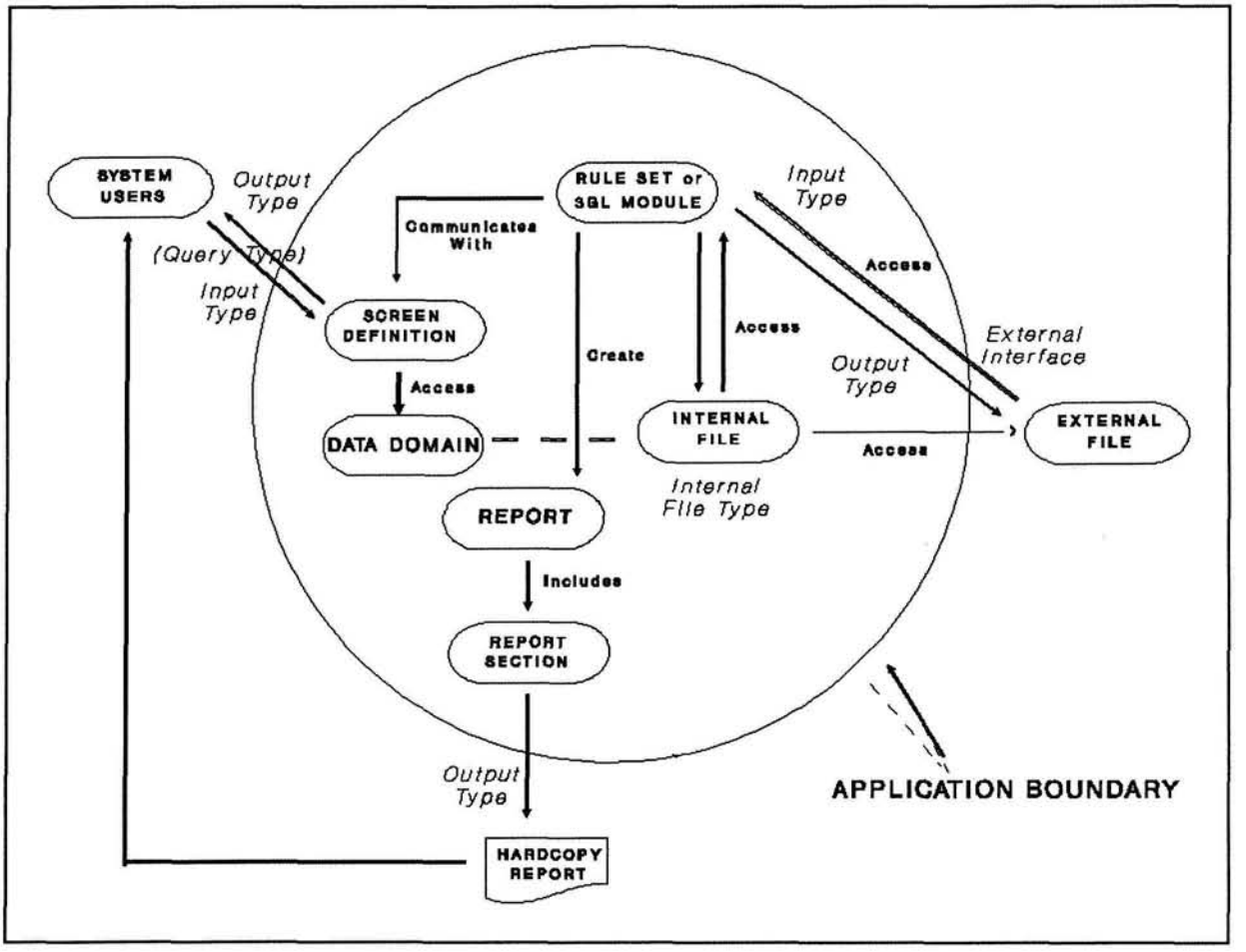

A BUSINESS FUNCTION is represented in ICE by a menu of BUSINESS PROCESSES. An application consists of all the objects called (directly or indirectly) by a given BUSINESS PROCESS. The first step in analyzing a system is to identify these objects, by iteratively tracing the calling relationships stored in the 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 (Figure 2). Note that the use of an object by an application system does not preclude its reuse by another application.

**FIGURE 2. ICE REPOSITORY OBJECTS**



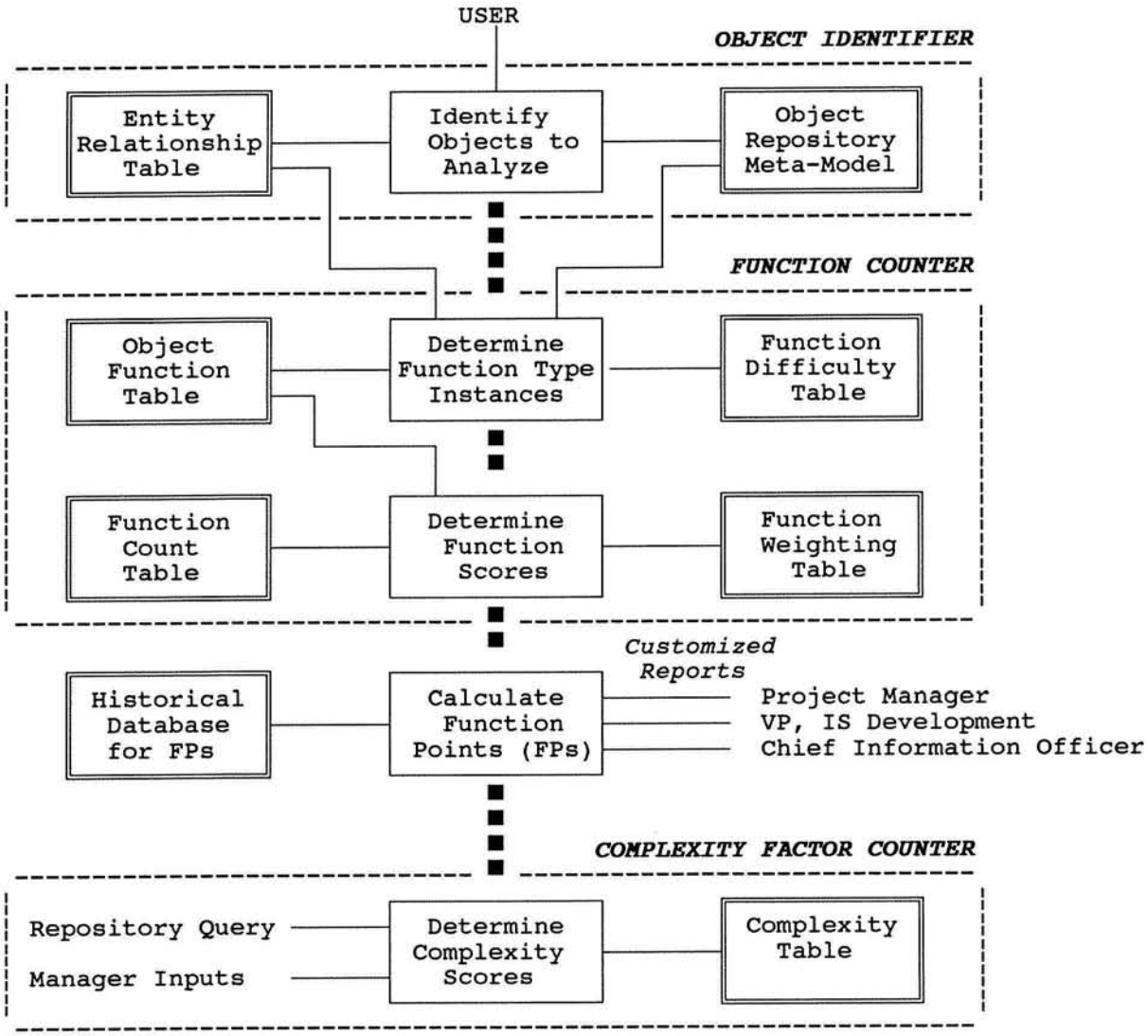

This figure is an expansion of RULE SET A, from Figure 1. There is a well-defined set of relationships allowed. Each object resides in the repository, and has a descriptive entry in a database table which also resides there. In addition, the repository contains other tables with entries for each relationship between two objects. A RULE SET may also use pre-existing 3GL MODULES. The repository contains no information about the processing performed by these modules. However, any functionality they provide the user, via REPORTS, FILES or SCREENS, must be mediated by an ICE object.

**FIGURE 3. MAPPING FROM ICE OBJECTS TO FUNCTION COUNTS**



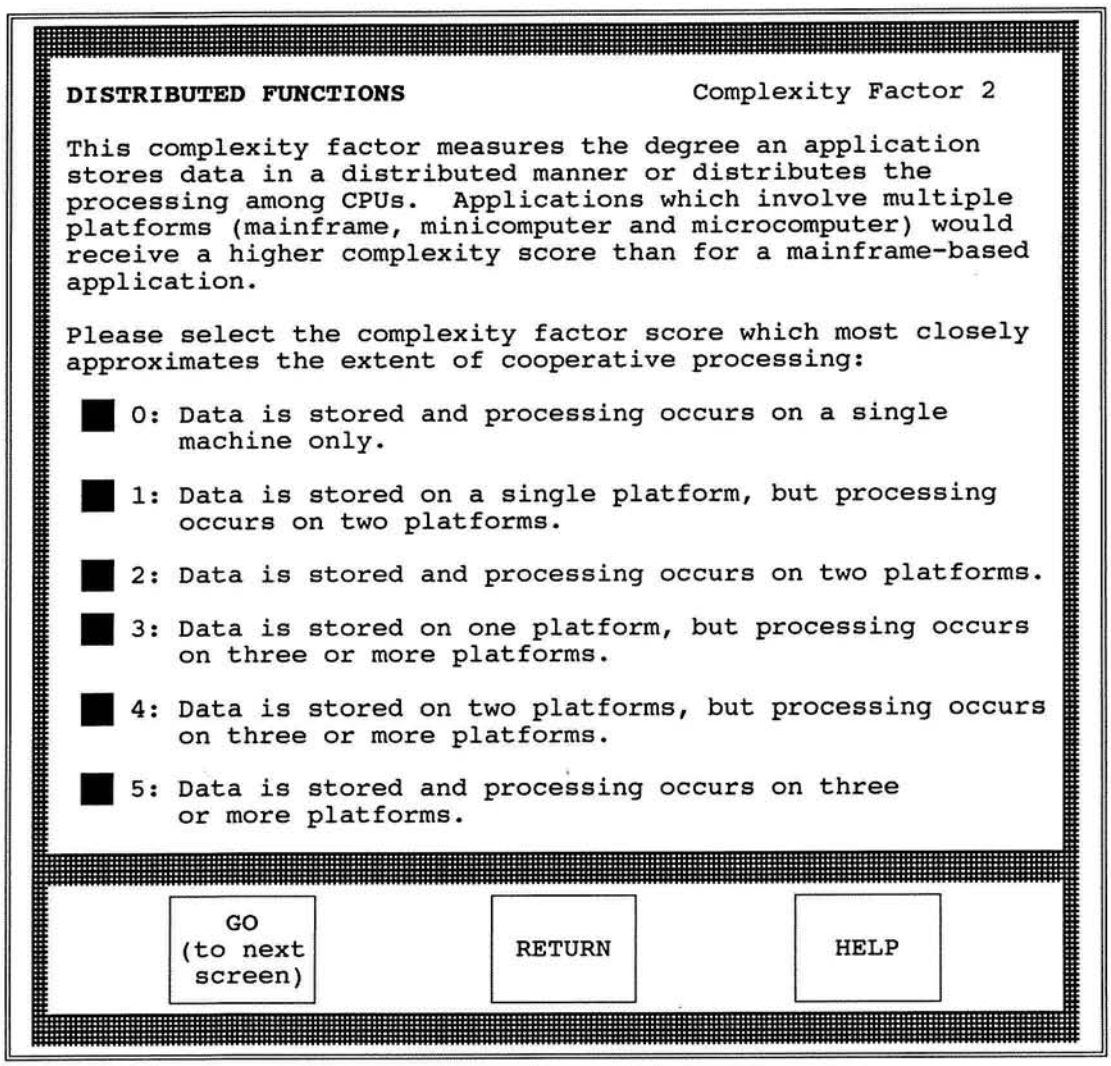

Function point analysis measures the functionality that a system delivers to the user in terms of data transfers into or out of that system (External Inputs, External Outputs, Queries, External Interfaces), and in terms of the data stores (Logical Internal Files) used. A 3GL program can contain functionality of all five classes. An ICE object, however, is severely constrained in the functionality it can represent, to the point where a system's function count can be computed by identifying and classifying its objects. See Table 1.

**FIGURE 4. THE AUTOMATED FUNCTION POINT ANALYZER: A SCHEMATIC**



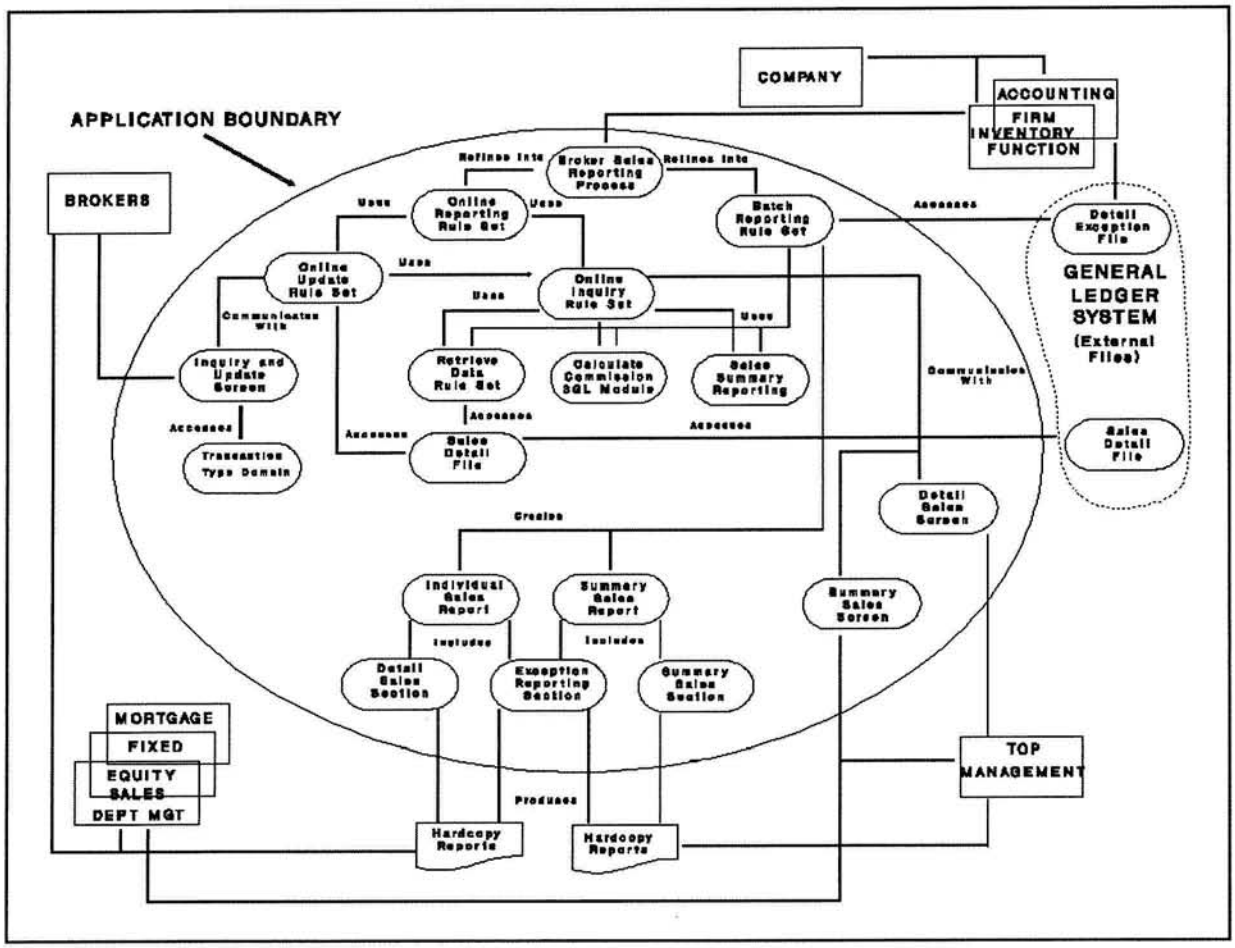

The Function Point Analyzer consists of three subsystems. One uses the meta-model to identify the objects in the application under analysis. The second uses it to assign Function Count scores to those objects. The third obtains task complexity measures (Table 3). This requires that the programmer or manager input in parallel with the automated analysis (Figure 5).

**FIGURE 5. FUNCTION POINT ANALYSIS COMPLEXITY MEASURES: AN INPUT SCREEN**




Each of the fourteen complexity factors of the function point methodology has its own input screen. Specific, objective descriptions, tailored to the organization's computing environment are given to anchor the scoring of the programmer or manager entering the data. Since some of the factors require human judgment, user input is still used in some cases. However, other complexity factors, such as the one above which measures the extent of distributed (or cooperative) processing, can be automated entirely, once the operational definition for this complexity factor has been implemented in terms of multi-platform processing and data flows using ICE, and validated by managers. At this time, such values are provided as modifiable defaults.

**FIGURE 6. THE BROKER SALES REPORTING SYSTEM: SYSTEM LAYOUT**




The Broker Sales Reporting System consists of those repository objects which are invoked by the Broker Sales Reporting Process, and of the relationships between those objects. The PROCESS *refines* into two RULE SETS, one for online processing and one for batch processing. Since the two RULE SETS generate similar outputs, they have a number of other repository objects in common. Each such object is only stored once in the repository, and reused as necessary. Each use will be instantiated in the meta-model as an entry in the table of relationships.

**FIGURE 7. A SUBSET OF THE BROKER SALES REPORTING SYSTEM**

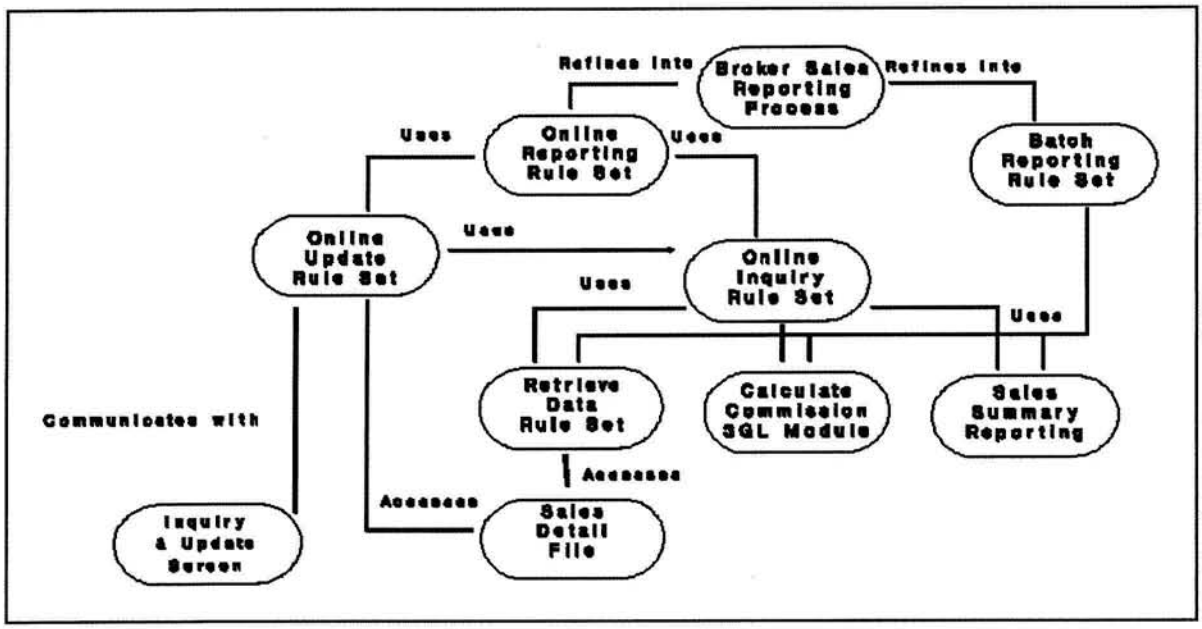



Figure 7 displays a subset of the Broker Sales Reporting System.

**FIGURE 8. EXPANDED HIERARCHY FOR A SUBSET OF THE BROKER SALES REPORTING SYSTEM**

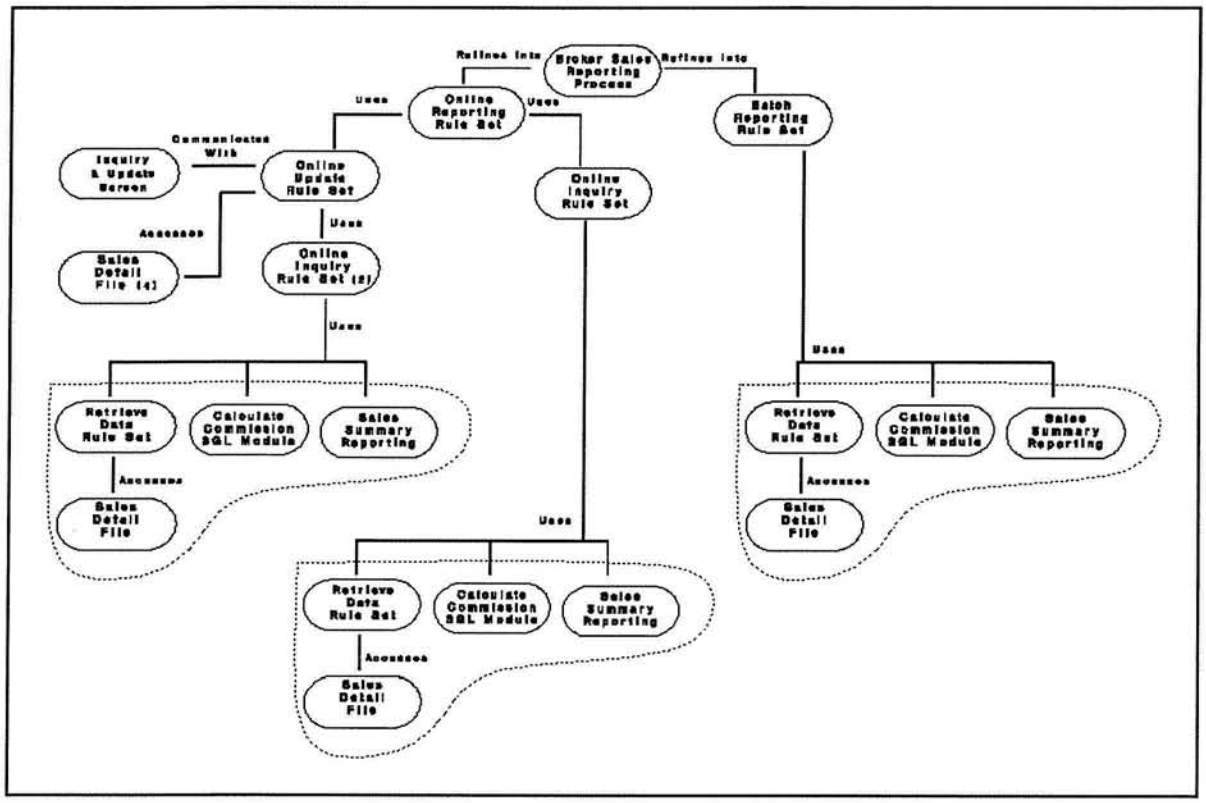



Figure 8 displays the same subset, as it would appear in the absence of code reuse. Several of the objects would have to be rewritten many times. Code Reuse Leverage is the ratio of the number of objects used (Figure 8) to the number of unique objects actually written for this application (Figure 7). The 3GL MODULE (Calculate Broker Commission) is *external* to this application; it was originally written for a different application, and reused by the programmers of this one. Therefore, the Code Reuse Analyzer will not include it in the count of unique objects written for this application.

**TABLE 1. FUNCTION POINT ANALYSIS FUNCTION COMPLEXITY MATRIX**

| FUNCTION TYPE | FUNCTION COMPLEXITY SCORES (c) | | |
|---|---|---|---|
| | **Simple** | **Average** | **Complex** |
| External Inputs | 3 | 4 | 6 |
| External Outputs | 4 | 5 | 7 |
| External Interfaces | 4 | 5 | 7 |
| External Queries | 4 | 5 | 6 or 7 |
| Internal Files | 7 | 10 | 15 |

The Function Point Analyzer can access a table of function count complexity measures which enable it to compute a function count score, once it has identified the mapping between ICE objects and the function types for a given application. The entries to the matrix above are the "standard" complexity measures of the function point analysis methodology, rather than calibrated measures relating to a specific CASE-development environment [63, 64].

**TABLE 2. REPOSITORY OBJECTS AND THE COMPUTATION OF FUNCTION COUNTS**

| OBJECT TYPES | BROKER SALES OBJECT NAMES | FUNCTIONALITY | FUNC-TION COUNT | TIMES USED | TOTAL COUNT |
|---|---|---|---|---|---|
| RULE SETS | On-line Reporting | | | 1 | |
| | Batch Reporting | Simple INPUT | 3 | 1 | 3 |
| | | Simple EXT. INTERFACE | 7 | 1 | 7 |
| | On-line Update | | | 1 | |
| | On-line Inquiry | | | 2 | |
| | Sales Retrieval | | | 3 | |
| | Sales Summary | | | 3 | |
| 3GL MOD-ULES | Calculate Commission | | | 3 | |
| REPORTS | Individual Sales | | | 1 | |
| | Summary Sales | | | 1 | |
| REPORT SEC-TIONS | Transaction Detail | Average Output | 5 | 1 | 5 |
| | Exception Reporting | Simple Output | 4 | 2 | 8 |
| | Summary | Average Output | 5 | 1 | 5 |
| SCREEN DEFIN-ITIONS | Detail Sales | 3 average QUERIES | 5 | 2 | 30 |
| | Summary Sales | 3 average QUERIES | 5 | 2 | 30 |
| | Inquiry and Update | Average INPUT | 4 | 1 | 4 |
| | | Average QUERY | 5 | 1 | 5 |
| | | Average OUTPUTS | 5 | 2 | 10 |
| DOMAINS | Transaction Types | Simple INTERNAL FILE | 7 | 1 | 7 |
| FILES | Transaction Detail | Average INTERNAL FILE | 10 | 4 | 40 |
| | | Average INPUT | 4 | 4 | 16 |
| | | Average EXT. INTERFACE | 7 | 4 | 28 |
| *TOTAL FUNCTION COUNT* | | | | | 198 |

*Note: For every screen which displays tabular data, ICE automatically generates a graphic-display screen and a HELP screen as well.*

The Function Point Analyzer identifies all the repository objects in the application system, and determines how many times each is used. The Detail Sales Screen, for example, is used twice: in response to an Online Inquiry and in response to an Online Update. In the latter case, the Online Update RULE SET *reuses* the Online Inquiry RULE SET and all the objects (including the Detail Sales Screen) which *it* uses.

The Analyzer then determines the function types associated with each object. An application's functionality depends upon its data stores and upon the flows of data (reports, queries, or updates) across its boundary. Thus almost all its function counts will be associated with REPORT SECTIONS, SCREENS or FILES. In this example, there is also some functionality associated with a RULE SET which has accessed a FILE belonging to a different application system.

**TABLE 3. COMPLEXITY MEASURES FOR THE BROKER SALES REPORTING SYSTEM**

| COMPLEXITY FACTOR | COMPLEXITY SCORE |
|---|---|
| Data Communications Requirements | 1 |
| Distributed Processing Requirements | 2 |
| Response Time or Performance Required | 1 |
| Heavily Used Configuration | 1 |
| High Transaction Rates | 2 |
| On-line Data Entry | 2 |
| End-User Efficiency | 2 |
| On-line Update | 3 |
| Complex Processing or Computations | 1 |
| Application Designed for Software Reuse | 3 |
| Application Designed for Ease of Installation | 3 |
| Application Designed for Ease of Operation | 3 |
| Application Designed for Multiple Sites | 2 |
| Application Designed to Facilitate Changes | 3 |
| TOTAL SCORE (Maximum possible is 70) | 29 |
| Adjustment Factor: (65+TOTAL SCORE)/100 | 0.94 |

The difficulty of developing an application depends not only on its magnitude (Function Counts) but also on the complexity of the tasks it performs. To adjust for this complexity, scores from 0 (no influence) to 5 (difficult) are assigned for each of fourteen factors. The resulting adjustment factor can modify the Function Count by up to 35% (plus or minus).

**TABLE 4. BROKER SALES REPORTING SYSTEM FUNCTION POINT SUMMARY**

| | |
|---|---|
| Number of Objects | 17 |
| Number of Function Types | 32 |
| Total Function Counts | 198 |
| Complexity Adjustment Factor | .94 |
| Total Function Points | 186 |

Function points are computed as the product of the Function Counts and the Complexity Adjustment Factor.

**TABLE 5. INSTANCES OF REUSE**

| Broker Sales Repository Object Name | Objects Written | Total Used | Object Complexity | Required Person-Days | Total Person-Days |
|---|---|---|---|---|---|
| Reporting Process | 1 | 1 | Simple | 2 | 2 |
| On-line Reporting Rule | 1 | 1 | Simple | 2 | 2 |
| Batch Reporting Rule | 1 | 1 | Simple | 2 | 2 |
| On-line Update Rule | 1 | 1 | Average | 4 | 4 |
| On-line Inquiry Rule | 1 | 2 | Simple | 2 | 4 |
| Sales Retrieval Rule | 1 | 3 | Average | 4 | 12 |
| Sales Summary Rule | 1 | 3 | Simple | 2 | 6 |
| Transaction Detail File | 1 | 4 | Simple | 3 | 12 |
| Transaction Type Domain | 1 | 1 | Simple | 1 | 1 |
| Compute Commission | EXT | 3 | Complex | (7) | 21 |
| TOTALS | 9 | 20 | | 22 | 66 |

The repository contains enough information for the automated Software Reuse Analyzer to classify each object as Simple, Average or Complex, on the basis of estimation heuristics used by ICE developers. (This is not the same classification used by the Function Point Analyzer.) These heuristics also enable the Analyzer to assign a programming-time estimate to each object, based on its type and complexity. Thus we can estimate the programming time required, and the programming time that would have been required in the absence of software reuse.

**TABLE 6. SOFTWARE REUSE METRICS**

| | | | | |
|---|---|---|---|---|
| **Leverage Metrics:** | | | | |
| Total number of objects used | 20 | | | |
| Number of unique objects written | 9 | | | |
| *New Object Percent* (9/20) | 2.2 | | | |
| *Reuse Leverage* (20/9) | 45% | | | |
| **Value Metrics:** | | | | |
| Total person-days of objects used | 66 | | | |
| Person-Days required for objects written | 22 | | | |
| *Object Reuse Value* (1-(22/66)) | 67% | | | |
| *Function Point Reuse Value* | Not implemented. | | | |
| **Classification Metrics:** | *Objects* | | *Person-Days* | |
| Unique objects written | 9 | 45% | 22 | 33% |
| *Reuse of internal objects* | 8 | 40% | 23 | 35% |
| *Reuse of external objects* | 3 | 15% | 21 | 32% |
| Total number of objects used | 20 | 100% | 66 | 100% |

On the average, each object is used 2.2 times. However, we see from the *reuse value* metric that without reuse the project would have taken approximately *three* times as long to write. The simple *leverage* metric underestimates the benefits of reuse in this case, because it does not distinguish that the more expensive objects are receiving a disproportionate amount of reuse.