AUTOMATING SOFTWARE DEVELOPMENT PRODUCTIVITY METRICS

Rajiv D. Banker
Eric Fisher
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

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

AUTOMATING SOFTWARE DEVELOPMENT PRODUCTIVITY METRICS

by

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

Eric Fisher
Doctoral Program in Information Systems
New York University

Robert J. Kauffman
Assistant Professor of Information Systems
New York University

Charles Wright
Seer Technologies

Dani Zweig
Assistant Professor of Information Systems
Naval Post Graduate School

September 1990

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

Working Paper Series

STERN IS-90-15

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 a large investment bank. We also appreciated the assistance and advice of Donna Dodson, Len Erlihk, Don Middleton, Norman Shing and Brian Weisinger in understanding the computer aided software engineering tools and development methods utilized within the industry. Finally, we wish to thank Hank Lucas and Vivek Wadhwa for invaluable suggestions about the managerial and technical content of this paper.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

AUTOMATING SOFTWARE DEVELOPMENT PRODUCTIVITY METRICS

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 object-oriented, integrated CASE -- have the potential to support the automation of this measurement. In this paper, we describe automated analyzers for function point and code reuse measurement. Both analyzers take advantage of the existence of a meta-model of the application system, stored within an object repository, which contains the necessary information *about* the application system. We also propose new metrics for code reuse analysis, including *reuse leverage*, *reuse value* and *reuse classification*. The state-of-the-art automated software metrics analyzers are illustrated in the context of an investment banking industry application.

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

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

1. Introduction

1.1. The Incentive and Opportunity to Automate Software Metrics

The recent upsurge in interest concerning computer aided software engineering (**CASE**) technologies 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."*([34], pp. D1-2)

Many observers believe this is a "software industrial revolution" in the making. However, the cost of participating in this revolution may be substantial, while the benefits have proven hard to verify [8, 31].

The opportunity is that of automating the collection of productivity data. Any firm with high software expenditures that is attempting to achieve important strategic and operational goals has a strong incentive to measure its productivity [9, 15, 33, 37]. But in traditional software shops, such measurement requires discouragingly expensive manual analysis of the software. CASE technologies, especially object-oriented, repository-based integrated CASE technologies, provide a means to automate a variety of software metrics that can help managers to gain control of their software development operations.[1]

A software development metrics expert recently commented that 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.[2] *Software Magazine* expressed a similar view of the future by showcasing products from nearly forty vendors which "measure productivity within a CASE environment" [8]. But a cursory

---

[1]For an introduction to the "repository" concept, see [12] and [17].

[2]From a speech by Capers Jones on software maintenance given at Center for Research on Information Systems, Stern School of Business, New York University, December 1989.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

review of the listing of products identified very few which actually automate the process of collecting software metrics to perform 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, however, is precisely what has made productivity measurement so difficult to carry out in the past.

In this paper we will examine the automation of two important metrics: *function points* -- a measure of programmer output -- and *code reuse* -- a major determinant of programmer productivity. Function point analysis is currently the most popular means of measuring the output of software development activities, although the analysis is quite labor intensive, especially for large systems. Code reuse is the extent to which software is developed by recycling previously written code rather than rewriting it from scratch. Extensive code reuse can increase productivity by an order of magnitude and more [3]. These two measures, which we will discuss in greater depth below, offer useful indicators of the productivity of software project performance in CASE development environments.

In order 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, "object-oriented programming" can provide this capability, primarily by encouraging the division of software into smaller and more easily analyzed units than the traditional program.[3] (For additional information on object-oriented development environments, the interested reader is referred to: [7, 10, 14, 27, 28, 29, 40].)

A prerequisite for any "industrial revolution in the making", is the ability to measure such basic factors as output and productivity. Yet, despite annual software costs rising into the hundreds of billions of dollars, and a general agreement that these costs are out of control, such measurement has generally proven too difficult and expensive to undertake. 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 bring it under control.

---

[3]A recent article in the New York Times provided a useful and readily understood definition of object-oriented programming: "In object-oriented programming, the data and the instructions are combined into a single module of software, or object ... Objects pass messages to one another requesting information and giving instructions. Yet no object interferes with the internal working of another. This method makes it easier to reuse pieces of software and to make changes" ([34], pp. D1-2).

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

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, object-oriented *Integrated CASE Environment* (which we will call "ICE" hereafter). These include a *Function Point Analyzer* (**FPA**) and a *Code Reuse Analyzer* (**CRA**).

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-based, object-oriented CASE development environments which enable us to automate function point analysis.

Section 3 presents 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.[4] 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 Code Reuse Analyzer. We define three classes of code reuse metrics, and discuss the design of CRA, and the manner in which it navigates the meta-model hierarchy to obtain the relevant information to instantiate the code reuse metrics.

The concluding section addresses additional technical and managerial questions that were raised by our work in this area, and the future research which is required to resolve them. It also summarizes the key contributions of this work to practitioners and to research on software development productivity. The paper also includes a stand-alone example of how the analyzers and the new metrics we will propose can be applied to an investment banking application called the Broker Sales Reporting System.

---

[4]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" ([12], p. 47). In this paper, we focus almost exclusively on the capability of a repository to store information concerning the relationship among objects which comprise a system.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

2. Automating Function Point Analysis: Preliminaries

2.1. Function Point Analysis

The magnitude of a software development effort depends upon several factors, including the amount of information processing accomplished by the system, the quality and the extent of the input and output interfaces provided to meet the users' needs, and environmental factors ranging from the quality of the hardware used by the programmers to the sophistication of the users requesting the software [42]. Allan Albrecht of IBM originally proposed "function points" as a metric to capture the intrinsic size of an application, so that software development activities could be evaluated for the *outputs* they create, and so that software development managers would have a tool to estimate the resources required to build systems of various sizes [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 output measures (such as source lines of code) that 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, into a well-accepted and operationally well-defined methodology [11, 44].[5]

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 summing the point scores which are assigned (on the basis of their complexity) to each input, output, internal file, external interface and query that comprising the system. Function counts are further adjusted by a measure of *environmental complexity*. The mathematical definition of *function points* is shown below.

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

where

FUNCTION COUNTS = instances of the five function types;

---

[5] For additional details on the implementation of function points which extends the approaches presented by Albrecht and Gaffney [2] and Zwanzig [44], see Symons [41], who discusses function points with entity type complexity rules.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

$COMPLEXITY_f$ = a complexity factor, f, associated with a descriptor of the implementation complexity of a system.

Two recent papers provide excellent critiques of function point analysis, alternative definitions and the issues that arise in calculating and using them [24, 42].

One roadblock to collecting function point metrics for software applications is that their computation (which is performed manually) is very labor-intensive. In addition, such computation requires the availability of consistently good system documentation. In practice, where documentation exists at all, it usually describes the system that was designed, not the system that was actually delivered.

A third concern is *calibrating* the people who carry out the function point analysis. Our experience in a recent 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 [3]. Individual differences in interpretation of documentation, knowledge of an application and experience in conducting function point analysis can all drive these differences. The recent research by Low and Jeffrey [24] 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.

2.2. ICE -- A Repository-Based, Object-Oriented Integrated CASE Tool

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

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

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

sets of highly skilled programmers.

ICE systems are written in an object-oriented fourth-generation language which buffers programmers from the complexity of the firm's operating environment. Applications written using ICE are later compiled in the appropriate languages for the relevant hardware platforms, and communication protocols for cooperative processing across platforms are handled without programmer intervention.

The object orientation of ICE supports cooperative processing. The code is organized according to objects which 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 additional 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 many-fold, 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 code 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, INSTANCES WITHIN 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, so as to enforce structured programming and design practices. We next consider each object type in more detail.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

A RULE SET contains most of the instructions which observers unfamiliar with CASE tools would tend to think of as "the program". In particular, most of the "traffic control" resides there: a RULE SET can *use* other RULE SETS or 3GL MODULES, *create* REPORTS which *include* REPORT SECTIONS, *access* FILES and *communicate with* SCREEN DEFINITIONS. (Note that the 4GL used by ICE has 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 third-generation language. While 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 is more labor-intensive by a full order of magnitude.

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 management, and it makes all the objects of one application available for reuse by any other application which is stored in the

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

repository.[6]

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. (See Figure 1.) A single application is invoked by a menu item which has 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.[7]

These relationships, like the objects themselves, reside in the object repository. Every such relationship is represented by a database entry, and collectively, this database of relationships constitutes the application "META-MODEL" -- the abstract structural map of the application system. (See Figure 2.)

We can use this general META-MODEL to identify the entities 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 (code 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.

---

[6]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" ([43], p. 229).

[7]The verbs in the ICE 4GL language 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 ICE enforces.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

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 code by multiple applications.

Second, the design of ICE's object-oriented 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. 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." (See Figure 3.)

3.1.1 INPUTS

A SCREEN with an output DATA VIEW (i.e., a SCREEN which sends data, as well as receiving it) is an input. A FILE access is an input if the FILE is external to the system. The complexity of the 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. OUTPUTS

A SCREEN with an input DATA VIEW (i.e., a SCREEN which receives data from the RULE SET which calls it) is an output, as is a REPORT or an output to an external FILE. Again, the complexity of the 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

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

(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. INTERNAL FILES

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 an 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 internal FILES. DATA DOMAINS are used by an application to validate or verify the values a user inputs.

3.1.5. EXTERNAL INTERFACES

A FILE that is not internal is considered to be external. Each occurrence of an external FILE access constitutes an external interface, as well as either an input or an 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 (see Table 2) 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.

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

FPA has three main components that execute the function point analysis methodology: an Object Identifier, a Function Counter and a Complexity Factor Counter. (See Figure 4.)

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

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

until all relevant objects have been found.

The *Function Counter* performs the mapping described in the previous section from objects and relations, 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 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.) 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 adjustment factor, according to the relation presented in Section 2.1.

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 environmental complexity. This data, along with the total function points and other useful managerial information can be tracked for completed systems, as well as for systems that are under construction. (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, 2, 3 and 4

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

at the end of this paper.)

4. CRA: A Code Reuse Analyzer for ICE

Code reuse is known to be a major source of productivity gains and cost reduction in software development operations. [32, 38]. A study conducted at the Missile Systems Division of the Raytheon Company found that greater than 60% of procedural code was repeated in multiple applications [6], 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 which are pervasively reported in the popular press, reuse represents a source of savings that managers are increasingly interested in tapping.

Yet, due to the difficulties associated with identifying reuse in most 3GL and 4GL environments, efforts to implement and manage successful code reuse programs have been stymied in many organizations [18, 25]. Traditionally, assessing the level of code reuse in a 3GL programming environment has been difficult. While 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-oriented CASE environment provides two major aids to the implementation and control of code reuse. *First*, the code exists at a level of granularity more conducive to the implementation of code reuse. While it is rare that an entire 3GL program will prove reusable, such programs frequently contain routines which *could* be reused with a little modification, were the programmer aware of their existence. An object-oriented system may be designed so that each such routine is a unique object. This makes reuse opportunities considerably easier to identify and to exploit.

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

To provide managers with information on code reuse, we designed and developed a facility within ICE called the Code Reuse Analyzer (**CRA**). CRA analyzes an existing software application, reporting the levels of reuse for the various elements comprising the application.

Like FPA, CRA identifies all the relevant objects for a given analysis by systematically navigating the hierarchy of calling relationships within the repository. (In fact, it reuses much of

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

the code from FPA.) Once all the objects within an application have been identified, and the instances of reuse have been noted, a range of managerially useful code reuse metrics can be computed.

4.1. Prior Research on Reuse

Prior research provides relatively little guidance as to how code reuse metrics should be defined. The bulk of the research concerning reuse in 3GL and 4GL environments has focused on exploiting the technology available to increase the level of reuse, rather than upon the impact of reuse on development productivity, and costs have not been considered. For example, Horowitz and Munson [18] looked at reuse in the context of compiler technology and subroutine libraries, application generators, and the development and adaptation of unspecialized systems which may be reused with minimal incremental development effort.

Other studies have made special efforts to define the range of possible kinds of reuse. For example, Jones [19] suggested the following kinds of reuse in software development operations: data, architecture, designs, programs and common subsystems and modules. Kernighan [21] 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. Still other researchers have explored how to promote reuse by suggesting new development methods, such as the "reusable module design" approach of Lanergan and Grasso [22], and "range-of-change requirements specification" of Matsumoto [26]. Our focus is limited to reuse of code, although ICE stores information about the functional and technical design of a system as well.

Two studies we identified made concrete suggestions regarding strategies for the measurement of reuse: Standish [39] and Neighbors [30]. Standish's proposal -- that re-use should be measured at the line of code level -- suffers from the disadvantages endemic to source-line-of-code metrics: they are conceptually simple, but are unlikely to convey managerially useful information. Neighbors argued that reuse should be abstracted from the level of source code into some meta-language which relates to the problem. This approach is likely to be of practical use in an environment in which a high-level representation of the system already resides.

Gaffney and Durek [13] modeled the cost impact of code reuse as a function of the relative costs of new and recycled code (there are costs associated with reuse; they are just usually lower than the cost of rewriting), and of their relative incidence. The quality of the authors' analysis suggests a strong rationale for creating code reuse metrics which support economic modeling of software development productivity and measurement of the business value of CASE technology. (For discussions of the value and use of

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

economics-based approaches to the evaluation of software development performance, see Boehm [5], Kang and Levy [20], and Levy [23].)

In the next section, we build on this discussion of the generally neglected problem of *measuring* code reuse.

4.2. Basic Definitions of Reuse

Since most studies of reuse have concentrated on the problems of encouraging it, rather than on those of identifying and measuring it, it is not surprising that there are few rigorous definitions of reuse in a systems development context [22, 32, 36]. *Reuse*, as the name implies, is the employment of previously written code as an alternative to writing new, possibly identical, code to perform the same or similar function.

The level of code reuse may be computed as the number of times a particular piece of code, data element or object is reused within the context of a program, application or information system [35]. As Hall [16] has pointed out, however, this measure does not, in itself, address many of the managerial questions concerning code 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?* (p. 41)

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

* What objects are being reused?

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

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

As a result, we have developed metrics to address all three kinds of questions: *reuse leverage metrics*, *reuse value metrics*, and *reuse classification metrics*, respectively. (For a fuller presentation of these ideas, see [4].)

4.2.1. Leverage Metrics

Reuse leverage metrics measure the number of times objects are

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

reused within a system. We define the degree of reuse within a system as:

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

For example, if a system consists of 400 objects, of which 100 had to be programmed from scratch, the reuse leverage would be 400/100 = 4.0, meaning that the average object is used 4 times. This measure of reuse can be used 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 4.0 aggregates a reuse leverage factor of 2.5 for RULES and 6.0 for SCREEN DEFINITIONS.

4.2.2. Value Metrics

To measure the actual productivity gains associated with code reuse, we must also distinguish between the reuse of easily-programmed objects, such as REPORTS and the reuse of more costly objects, such as RULE SETS. We can compute *reuse value* by weighting the level of reuse by the cost of programming the various types of objects. Specifically, rather than just counting objects, we add up the *cost* of each object:

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

where

$COST_j$ = the standard (or average) cost in person days of building object j;

J = the total number of occurrences of objects in an application META-MODEL hierarchy;

k = the total number of *unique* objects built for this application.

This metric provides an estimate of the percentage of development costs saved, assuming the calculation of total costs is made based on the standard costs associated with the various object types.

4.2.3. Classification Metrics

For most purposes we include in our computation of code reuse, any object which is found in the repository, rather than rewritten from scratch. For some managerial purposes, however, we will wish to

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

distinguish *internal reuse* from *external reuse*. Internal reuse refers to code reuse within a system or subsystem, as defined by its META-MODEL hierarchy. (For example, almost all the reuse displayed in Figures 7 and 8 is of this type.) External reuse refers to the reuse of objects which are in the repository, but which currently belong to a different system, and were originally developed for it. While both kinds of reuse are of equal value, different managerial policies may be required to encourage the two kinds of reuse.

In particular, the degree of internal reuse will probably depend upon the size of the team developing a given application, and the quality of the communications within that team. The degree of external reuse, on the other hand, will depend more upon the quality of the indexing system used to help programmers to identify existing objects which they might be able to reuse. When reuse metrics are being computed for all the code within the repository, all reuse is internal.

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

$$\textit{NEW CODE PCT} = \frac{\textit{NUMBER OF NEW OBJECTS BUILT}}{\textit{TOTAL NUMBER OF OBJECTS USED}}$$

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

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

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.

4.3. CRA Architecture

The Code Reuse Analyzer 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 its objects, and CRA uses it to trace all the objects which are called, directly or indirectly, by the application under analysis. As for 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 large or small set of

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

application systems, or even a subset of a single system.

The ability to start anywhere in the hierarchy provides CRA 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.

Once the set of objects has been identified, the objects can be classified, and reuse metrics can be computed. 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 code, while subsequent encounters are classified as instances of internal reuse. (An illustration of how CRA calculates the code 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.)

5. Conclusion

We have described two automated software development productivity analyzers, a function point analyzer and a code reuse analyzer. In the process of building 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 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.

ICE, for example, automatically provides many capabilities which 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 (a useful capability for traders 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

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

the superior functionality and user-friendliness which may be expected to accompany CASE development.[8]

5.1. 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 Code Reuse Analyzer gave us an improved understanding of code reuse. In particular, we have confirmed that commercial application systems offer tremendous scope for code reuse. If the average object is used five times, this can mean an 80% reduction in the cost of programming and unit testing, and such reuse levels are attainable within ICE. However, initial analysis suggests that, even here, little of the potential for reuse is being tapped. In particular, programmers tend to only reuse code with which they are personally familiar.

We are now in the process of formulating research to deal with the questions raised by these observations. How can code reuse be supported, encouraged and motivated? What aspects of the code are conducive to reuse? What programming practices and what managerial practices provide the proper incentives for code reuse?

One of the major benefits of the development of the automated analyzers to our research efforts is the outputs they will create. The automated report generation capabilities of the Function Point and Code Reuse Analyzer enables us to pursue research questions that were simply beyond the scope (in terms of cost and availability of data) of prior research. The basic questions are: 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?

---

[8]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.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

The questions raised here are the basic questions the software development managers will have to answer: What works? What doesn't work? How well does it work? How can I make it work better? In the absence of the right metrics made available to managers at the right time, it will be impossible to match the "art" of software management to the new "science" of software development.

5.2. Contributions

This paper had multiple objectives. We wished to report on our automation of function point and code 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 which make this automation possible. And we wished to report on the implications which this research has for our understanding of software productivity in an integrated CASE environment.

The Function Point Analyzer and the Code Reuse Analyzer described in this paper represent the state-of-the-art in designing and developing automated software metrics facilities in an integrated CASE tool environment. Their implementation was made possible by two key features of the object-oriented, repository-based integrated CASE environment which we have called ICE.

The first of these features is the repository, which contains not only all the code 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 would or could be maintained in such circumstances.

The second feature is the object orientation of the CASE environment, and of 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 proposed metrics for assessing code 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 aggregate reuse, as well as reuse of individual objects that may not be equally easy 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 code reuse lacks the power to answer the

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

questions that we found to be important to managers.

We also showed how traversing a hierarchical meta-model of an object-oriented system enables the analyst to identify objects, and define reuse which is internal to the hierarchy (for example, code reused within a program or an application) or which is external to it. Initial analysis suggests that this classification is important to managers wishing to encourage code reuse [4]. It appears that internal reuse will proliferate where the technology supports it: ICE programmers routinely reuse code from one part of an application in another. Code external to the system, however, tends to be code written by other programmers, and different technical support and organizational incentives are needed in order to motivate programmers to seek out external reuse opportunities.

Clearly, these questions are only the starting point for a rich, new management agenda to better understand and control CASE-based development. Yet, we are also 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.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

REFERENCES

1. Albrecht, A. J., Measuring Application Development Productivity. In *Proceedings of the Joint SHARE, GUIDE, and IBM Application Development Symposium*, IBM (October 1979), pp. 83-92.

2. Albrecht, A. J. and Gaffney, J. E. Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering* 9, 6 (November 1983), pp. 639-647.

3. Banker, R. D., and Kauffman, R. J. An Empirical Assessment of Computer Aided Software Engineering (CASE) Technology, A Study of Productivity, Reuse and Functionality. *Management Information Systems Quarterly* (forthcoming).

4. Banker, R. D., Kauffman, R. J. and Zweig, D. Metrics for the Code Reuse in Software Development. Working paper, 1990.

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

6. Boehm, B. W., and Papaccio, P. N. Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering* 13, 10 (October 1988), pp. 1462-1477.

7. Booch, G. Object Oriented Development. *IEEE Transactions on Software Engineering* 12, 2 (February 1986), pp. 211-221.

8. Bouldin, Barbara M. CASE: Measuring Productivity -- What Are You Measuring? Why Are You Measuring It? *Software Magazine* 9, 10 (August 1989), pp. 30-39.

9. Davis, G. B. Commentary on Information Systems: Productivity Gains from Computer Aided Software Engineering. *Accounting Horizons* 2, 2 (June 1988), pp. 90-93.

10. Deutsch, L. P. Reusability in the Smalltalk-80 Programming System. In *ITT Proceedings on Reusability in Programming*, ITT (1983), pp. 72-76.

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

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

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

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

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

15. Grammas, G. W., and Klein, J. R. Software Productivity as a Strategic Variable. *Interfaces* 15, 3 (May-June 1985), pp. 116-126.

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

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

18. Horowitz, E. and Munson, John. An Expansive View of Reusable Software. *IEEE Transactions on Software Engineering* SE-10, 5 (September 1985), pp. 477-487.

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

20. Kang, K. C., and L. S. Levy. Software Methodology in the Harsh Light of Economics. *Information and Software Technology* 31, 5 (June 1989), pp. 239-249.

21. Kernighan, B. W. The UNIX System and Software Reusability. *IEEE Transactions on Software Engineering* SE-10, 5 (September 1984), pp. 513-518.

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

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

24. Low, G. C., and Jeffrey, D. R. Function Points in the Estimation and Evaluation of the Software Process. *IEEE Transactions on Software Engineering* 16, 1 (January 1, 1990), pp. 64-71.

25. Mathis, R. F. The Last 10 Percent. *IEEE Transactions on Software Engineering*, SE-12, 6 (June 1986), pp. 705-712.

26. Matsumoto, Y. Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels. *IEEE Transactions on Software Engineering*, SE-10, 5 (September 1984), pp. 502-512.

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

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

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

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

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

31. Norman, R. J., and Nunamaker, J. F. Jr. CASE Productivity Perceptions of Software Engineering Professionals. *Communications of the ACM* 32, 9 (September 1989), pp. 1102-1108.

32. Nunamaker, J. F. Jr., and Chen, M. Software Productivity: A Framework of Study and an Approach to Reusable Components. In *Proceedings of the 22nd Hawaii International Conference System Sciences*, IEEE (Kona Lua, Hawaii, January 1989a), pp. 959-968.

33. Nunamaker, J. F. Jr., and Chen, M. Software Productivity: Gaining Competitive Edges in an Information Society. In *Proceedings of the 22nd Hawaii International Conference on System Sciences*, IEEE (Kona Lua, Hawaii, January 1989b), pp. 957-958.

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

35. Polster, F. J. Reuse of Software Through Generation of Partial Systems. *IEEE Transactions on Software Engineering* SE-10, 5 (September 1984), pp. 402-416.

36. Raj, R. K. and Levy, H. M. A Compositional Model for Software Reuse. *The Computer Journal* 32, 4 (April 1989), pp. 312-323.

37. Senn, J. A. and Wynekoop, J. L. Computer Aided Software Engineering (CASE) in Perspective. Working Paper, Information Technology Management Center, College of Business Administration, Georgia State University, 1990.

38. Seppanen, V. Reusability in Software Engineering. In P. Freeman (ed.), *Tutorial: Software Reusability*, Computer Society of the IEEE, 1987, pp. 286-297.

39. Standish, T. A. An Essay on Software Reuse. *IEEE Transactions on Software Engineering* SE-10, 5 (September 1984), pp. 494-497.

40. Stroustrup, B. What is Object-Oriented Programming? *IEEE Software* (May 1988), pp. 10-20.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

41. Symons, C. R. *Extended Function Points with Entity Type Complexity Rules*. Nolan, Norton, and Company, London, 1984.

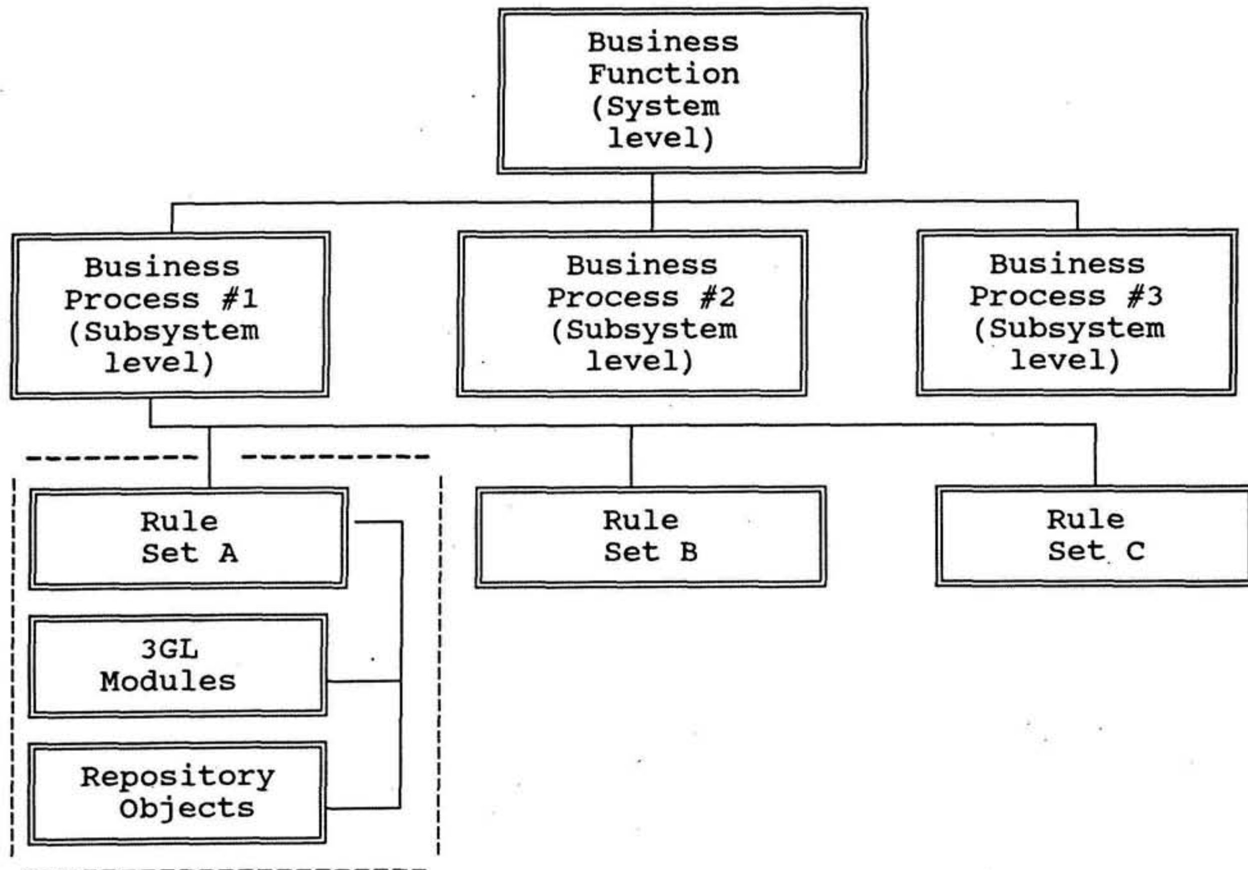
42. Symons, C. R. Function Point Analysis: Difficulties and Improvements. *IEEE Transactions on Software Engineering* 14, 1 (January 1988), pp. 2-10.

43. Veryard, R. Information and Software Economics. *Information and Software Technology* 31, 5 (June 1989), 226-230.

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

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

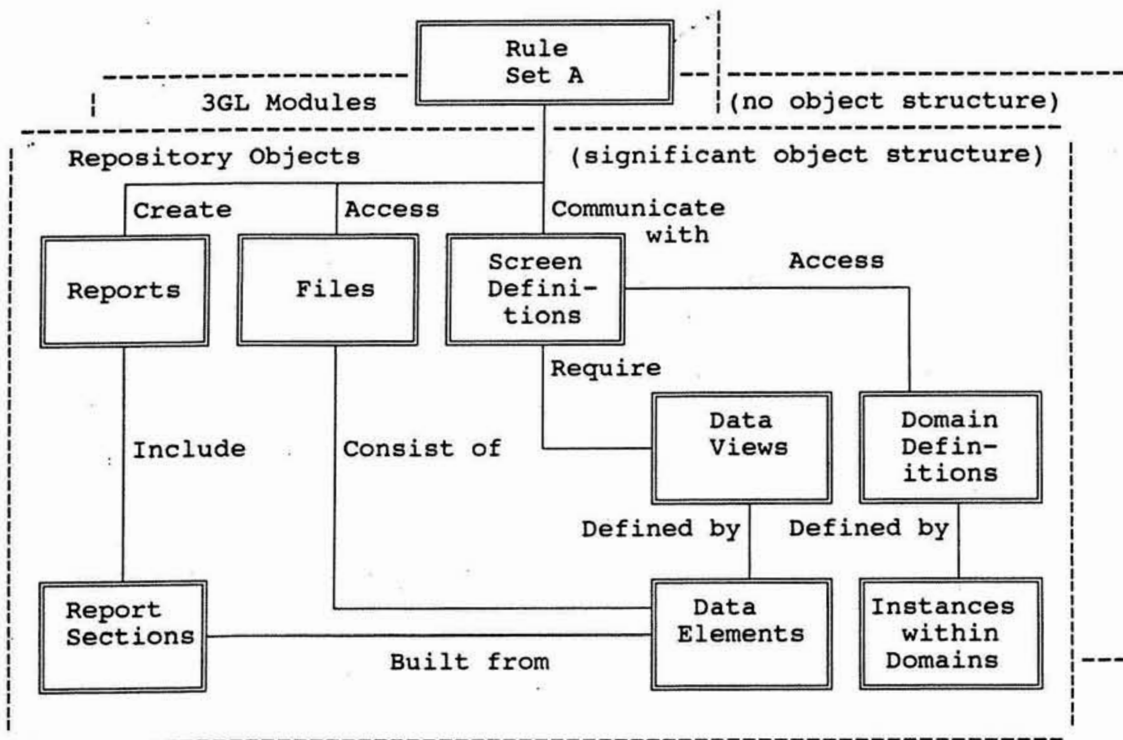
Figure 1. A Repository-Based Application Meta-Model



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.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

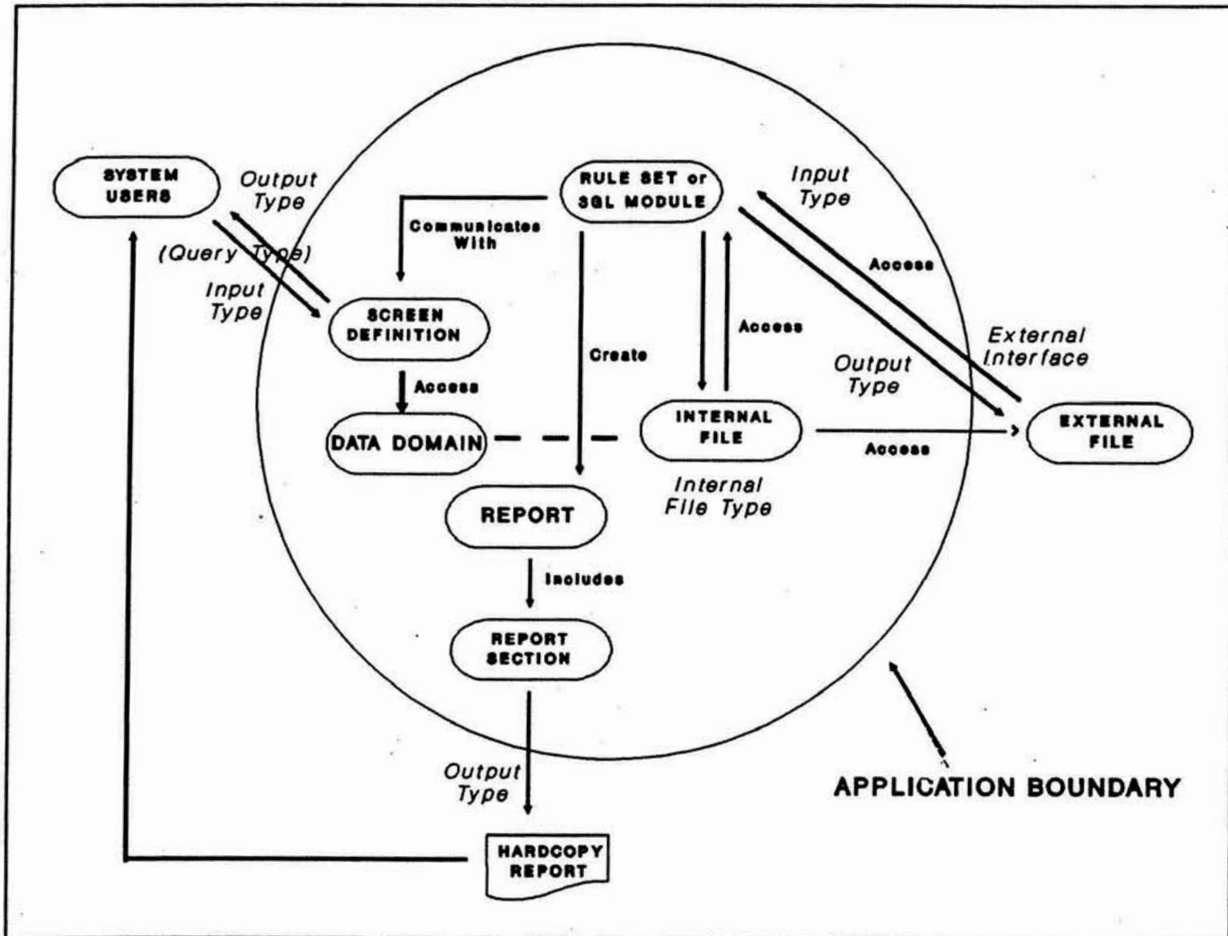
Figure 2. ICE Repository Objects



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.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

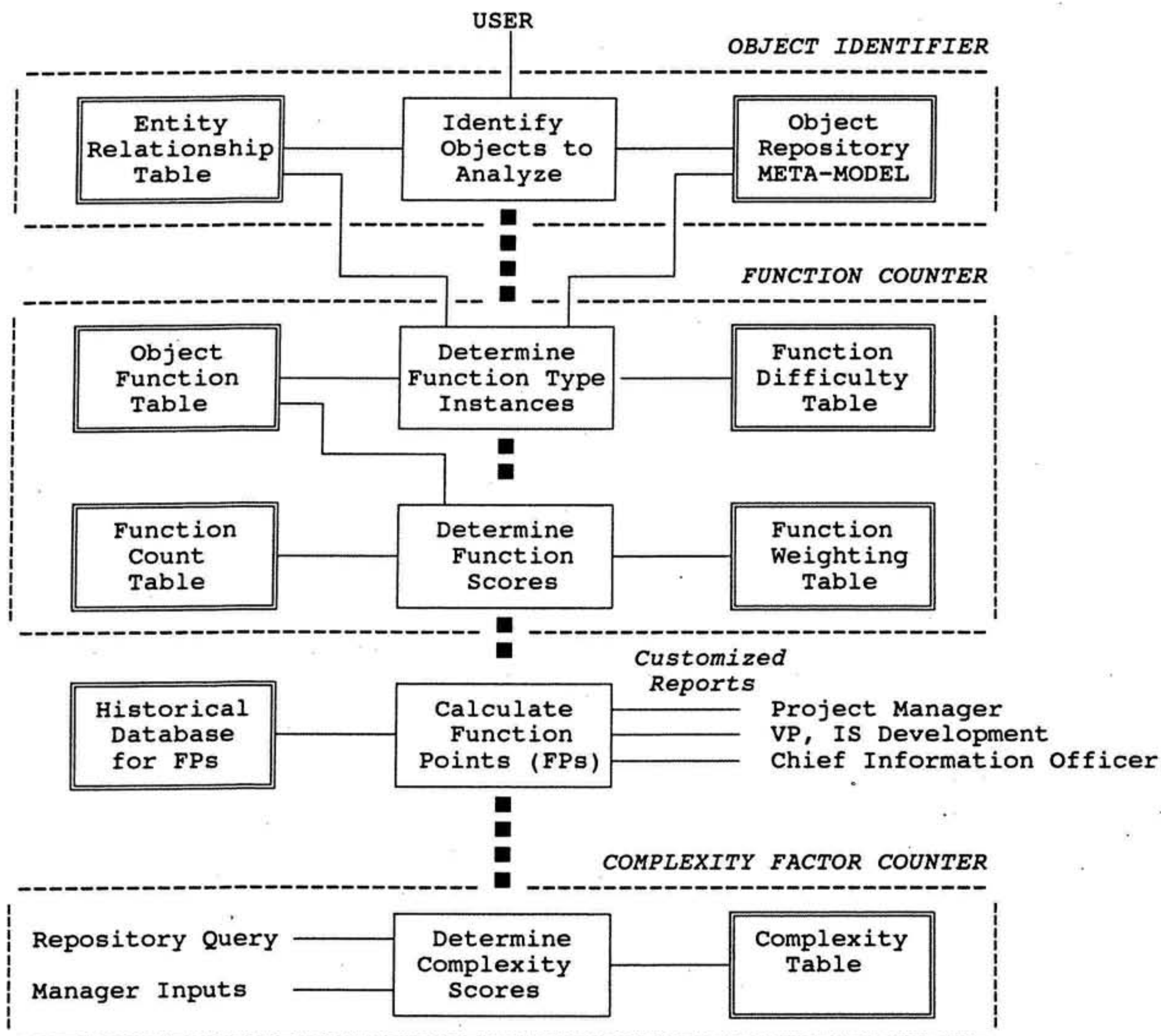
Figure 3. Mapping from ICE Objects to Function Counts



Function Point analysis measures the functionality which a system delivers to the user in terms of data transfers into or out of that system (Inputs, Outputs, Queries, External Interfaces), and in terms of the data stores (files) used. A 3GL program 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.).

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

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 programmer or manager input in parallel with the automated analysis (Figure 5).

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

Figure 5. Function Point Analysis Complexity Measures: An Input Screen

COOPERATIVE PROCESSING Complexity Factor 2

This complexity factor measures the degree an application stores data in a distributed manner or distributes the processing among CPUs. Applications which involve multiple platforms (mainframe, minicomputer and microcomputer) would receive a higher complexity score than for a mainframe-based application.

Please select the complexity factor score which most closely approximates the extent of cooperative processing:

- 0: Data is stored and processing occurs on a single machine only.
- 1: Data is stored on a single platform, but processing occurs on two platforms.
- 2: Data is stored and processing occurs on two platforms.
- 3: Data is stored on one platform, but processing occurs on three platforms.
- 4: Data is stored on two platforms, but processing occurs on three platforms.
- 5: Data is stored and processing occurs on three platforms.

GO (to next screen) | RETURN | HELP

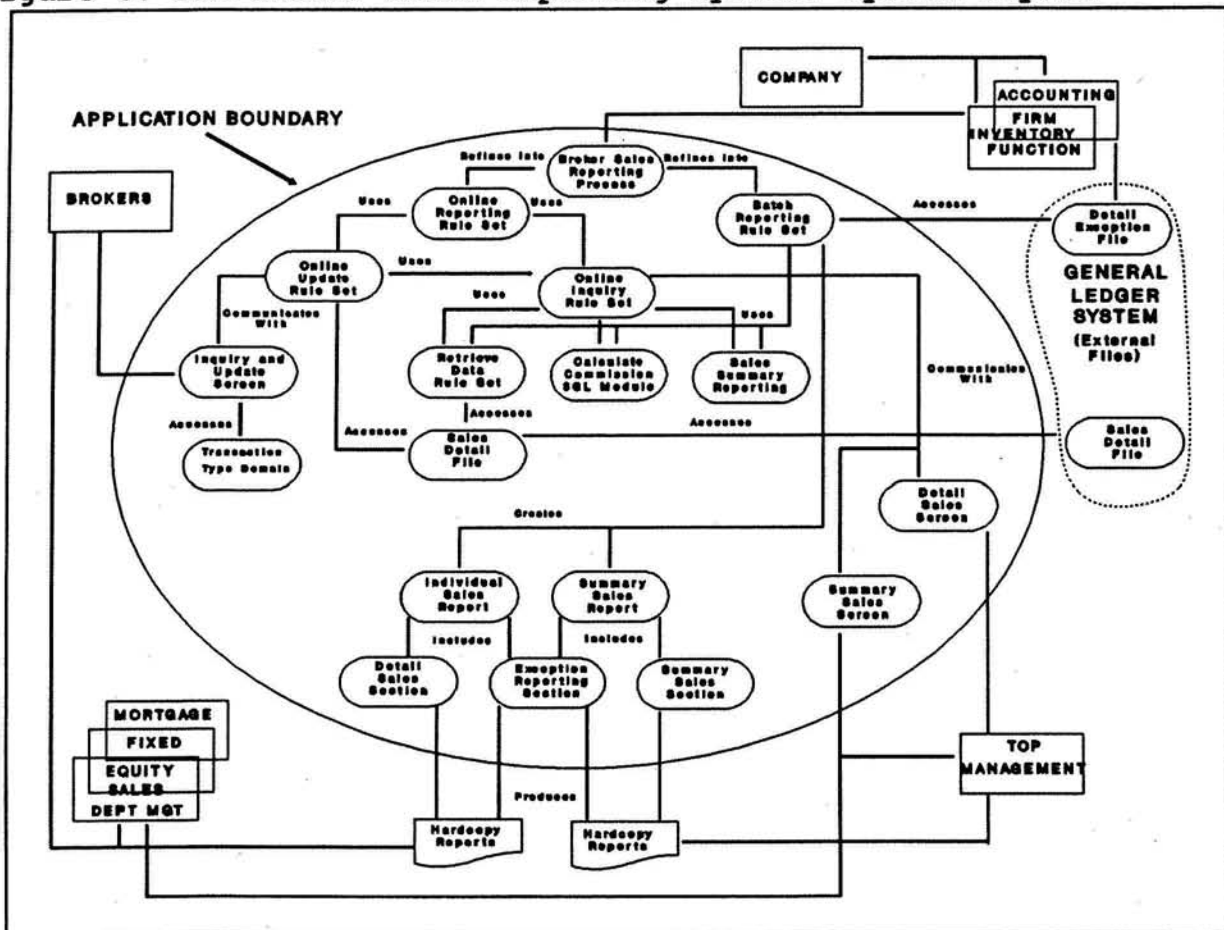
Each of the fourteen complexity factors has its own input screen. Specific, objective descriptions are given to anchor the scoring of the programmer or manager entering the data. Since some of the factors require human judgment, user input is still used in some cases. However, other complexity factors, such as the one above which measures the extent of cooperative (distributed) 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.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

Sidebar 1. The Broker Sales Reporting System: Introduction

The Broker Sales Reporting System is a small (simplified) ICE application system which 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 brokers.

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.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

Sidebar 2. The Code Reuse Analyzer

The operation of the Code Reuse Analyzer will be illustrated for a subset of the Broker Sales Reporting System. Code reuse is a measure of 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 Code Reuse Leverage. The Code Reuse Value metric estimates the *savings* attributable to reuse, by considering not only the number of objects, but also the cost of the objects.

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.[9]

The Code 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 code reuse goes unexploited.

---

[9]These heuristics are in actual use by managers, for project cost estimation. We are currently conducting additional research to determine their robustness.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

Figure 7. A Subset of the Broker Sales Reporting System (from Figure 3)

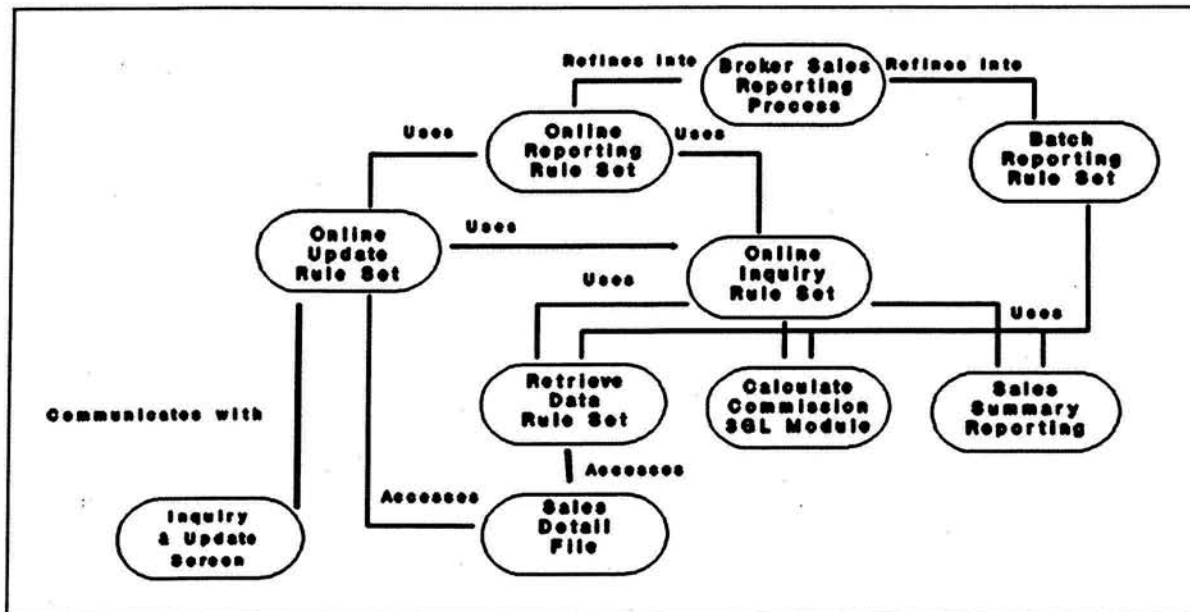


Figure 7 displays a subset of the Broker Sales Reporting System.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

Figure 8. Expanded Hierarchy for a Subset of the Broker Sales Reporting System

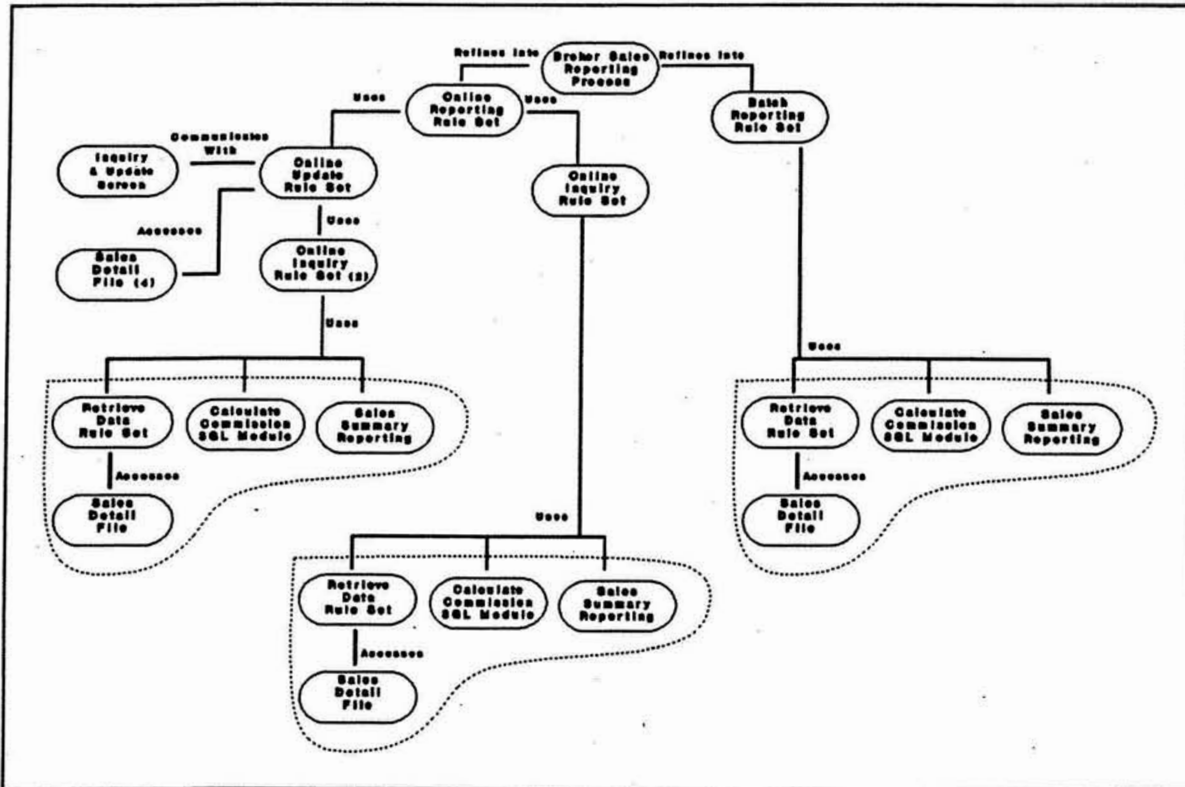


Figure 8 displays the same subset, as it would appear in the absence of code reuse. Several of the objects would have to be rewritten many times. Code Reuse Leverage is the ratio of the number of objects used (Figure 8) to the number of unique objects actually written for this application (Figure 7). 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. Repository Objects and the Computation of Function Counts

Object Type	Broker Sales Object Name	Functionality	Function Count	Times Used	Total Count
RULE SETS	Online Reporting			1	
	Batch Reporting	Simple INPUT	3	1	3
		Simple EXTERNAL INTERFACE	7	1	7
	Online Update			1	
	Online Inquiry			2	
	Sales Retrieval			3	
	Sales Summary			3	
3GL MODULES	Calculate Commission			3	
REPORTS	Individual Sales			1	
	Summary Sales			1	
REPORT SECTIONS	Transaction Detail	Average OUTPUT	5	1	5
	Exception Reporting	Simple OUTPUT	4	2	8
	Summary	Average OUTPUT	5	1	5
SCREEN DEFINI-TIONS	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 EXTERNAL INTERFACE	7	4	28
		Total Function Count			198

* 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.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

Table 2. Function Point Analysis Function Complexity Matrix

FUNCTION TYPE	FUNCTION COMPLEXITY SCORES		
	Simple	Average	Complex
External Inputs	3	4	6
External Outputs	4	5	7
External Interfaces	5	7	10
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 [11, 44].

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

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 Turnaround Time Required	1
Heavily Used Configuration	1
High Transaction Rates	2
Online Data Entry	2
Online Update	3
End-User Efficiency	3
Complex Processing or Computations	1
Code Designed for Reuse	3
Application Designed for Ease of Installation	3
Application Designed for Ease of Operation	4
Application Designed for Ease of Modification	2
Application Runs at Multiple Sites	5
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	x .94
Total Function Points	186

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

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

Table 5. Instances of Code Reuse

Broker Sales Repository Object Name	Objects Written	Total Used	Estimated Complexity	Required Man-Days	Total Man-Days
Reporting Process	1	1	Simple	2	2
Online Reporting Rule	1	1	Simple	2	2
Batch Reporting Rule	1	1	Simple	2	2
Online Update Rule	1	1	Average	4	4
Online 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 Code 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 code reuse.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15

Table 6. Code Reuse Metrics

REUSE LEVERAGE		
Total number of objects used		20
Number of unique objects written		9
Code Reuse Leverage (20/9)		2.2
REUSE VALUE		
Total Man-Days of objects used		66
Man-Days required for objects written		22
Code Reuse Value (1-(22/66))		67%
REUSE CLASSIFICATION	OBJECTS	MAN-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.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-90-15