MONITORING THE SOFTWARE ASSET: REPOSITORY EVALUATION OF SOFTWARE REUSE

Rajiv D. Banker
Robert J. Kauffman
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-91-32

MONITORING THE SOFTWARE ASSET: REPOSITORY EVALUATION OF SOFTWARE REUSE

by

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

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

and

Dani Zweig
Assistant Professor of Information Systems
Naval Postgraduate School

October, 1991

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

Working Paper Series

STERN IS-91-32

The authors wish to acknowledge Mark Baric, Gene Bedell, Gig Graham, Norm Leibson, Tom Lewis, Bob Menar, Vivek Wadhwa, and Jim Yent for the access they provided us to data on software development projects and managers' time throughout our field study of CASE development at Carter Hawley Hale, the First Boston Corporation and SEER Technologies. We also thank Michael Oara and Rachna Kumar for their assistance with the development and implementation of the repository evaluation queries. All errors in this paper are the responsibility of the authors.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

MONITORING THE SOFTWARE ASSET:

REPOSITORY EVALUATION OF SOFTWARE REUSE

ABSTRACT

Traditionally, software management has focused primarily upon cost control. Today, with the emerging capabilities of computer aided software engineering (**CASE**) and corresponding changes in the development process, the opportunity exists to view software development as an activity that creates reusable software assets, rather than just expenses, for the corporation. With this opportunity comes the need to monitor software at the corporate level, as well as at that of the individual software development project. Integrated CASE environments can support such monitoring. In this paper we propose the use of a new approach called *repository evaluation*, and illustrate it in an analysis of the evolving repository-based software assets of two large firms that have implemented integrated CASE development tools. The analysis shows that these tools have supported high levels of software reuse, but it also suggests that there remains considerable unexploited reuse potential. Our findings indicate that organizational changes will be required before the full potential of the new technology can be realized.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

1. INTRODUCTION

Traditionally, the management of software development has focused upon controlling its expense. The predicted benefits of a software development project may seem remote or speculative, but its costs are immediate, and highly visible. The resulting tendency to focus management attention on cost reduction is reinforced by the generally accepted practice of reporting software development as an expense, rather than as an investment (Boehm and Pappacio, 1988). As a result, the prime concern of senior management has been to limit current software development costs, and to control those that are likely to occur in the future when current systems are maintained and enhanced (Bailey and Basili, 1981; Grammas and Klein, 1985).

1.1. Monitoring Software Expenses

Following the time-worn epithet "You can't manage what you can't measure," it is clear that if management efforts are to focus on controlling software costs, then measurement efforts similarly must focus on:

* gauging a variety of dimensions of development costs and the resulting software outputs; and,
* determining the levers that can be used by management to improve cost efficiency.

To date, the bulk of management efforts to improve cost efficiency in software development have centered on improving software development project management. Much of this has been accomplished through the establishment of measurement and metrics programs that gauge project-level development productivity, for example, in terms of function points delivered per person month of development, or performance in terms of development cost overruns as a percentage of budgeted project costs.

We can take an alternative perspective, as well. This perspective involves thinking of software development as a process that leads to the creation of corporate assets that

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

will produce value for the firm over an extended lifetime of use. In this view, emerging computer-aided software engineering (CASE) tools that emphasize software reusability (Lenz, Schmid and Wolfe, 1987; Pollack, 1990; Rombach, 1991) can mean that much of the real value of modular "software assets" will be derived from the extent to which they can:

* defray the costs of the construction and testing, and raise the overall level of perceived quality and reliability of systems that are delivered;
* speed the implementation of new systems while first-mover competitive advantage opportunities still exist in the business areas that the software is meant to support;
* be leveraged across projects and areas of the firm in support of multiple businesses.

1.2. Repository Evaluation: A New Measurement Paradigm for Software Asset Management

The software asset management view was recently articulated by Banker and Kauffman (1991a) and Karimi (1990) and related issues are under scrutiny by a number of research groups (see for example, Apte, Sankar, Thakur, and Turner, 1990; Banker, Kauffman and Zweig, 1990; Barnes and Bollinger, 1991; Basili, 1991; Chen and Sibley, 1991; Nunamaker and Chen, 1989), whose investigations focus on how CASE tools change the software development process through software reusability in the design, construction, testing, and maintenance phases of the software development life cycle.

Where should measurement of software as an asset occur? Monitoring a system as it is built will answer many questions, especially questions concerning the factors affecting the cost of building it. To manage it as a contributing part of the software asset requires that we also be able to monitor it at the level of the organization or enterprise. Even relatively simple metrics, collected at that level, can answer key questions for senior managers -- questions that are important to long-term firm performance, but that have not been well articulated or that have been ignored altogether.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

This paper develops empirical results based on a new software engineering measurement paradigm: *repository evaluation*. An integrated CASE environment maintains all of its software and, more importantly, all of its information *about* that software (its design, its history, its interactions with other system elements) in a single structured repository. Much of this information is precisely the information that can support the management of those software assets.

The next section explores more deeply the issues that have led us to this perspective, and the kinds of questions that it will enable management to answer. Section 3 illustrates the measurement paradigm using data obtained from two large firms that have implemented repository-based integrated CASE tools. On the basis of an intuitively appealing model of software reuse, an analysis of the firms' repositories is used as a basis for evaluating their software reuse efforts. Section 4 presents additional discussion of software reuse within those firms, and uses the results of the analysis to examine the model in greater depth. The paper concludes by outlining our broadened understanding of software reuse, and by identifying new questions raised by our repository evaluation.

2. SOFTWARE ASSET MANAGEMENT AND REPOSITORY EVALUATION: THE NEED FOR NEW PARADIGMS

Our call for a new measurement paradigm is based on three related elements. *First*, there are important questions related to a firm's software development activities, which project-level analysis cannot adequately address, but that need to be examined more closely. *Second*, new software design paradigms are emerging -- object-based and object-oriented design in particular (Booch, 1989; Meyer, 1988) -- whose full benefits can only be realized through enterprise-level software management. *Third*, CASE technology, and especially repository-based integrated CASE, offers a mechanism for automating measurement so that more frequent and detailed data collection is feasible, without forcing management to shift labor away from development and into the manual collection of project performance metrics. This creates significant opportunities to study software

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

reusability.

2.1. Why Measure at Levels Other Than the Project Level?

At the core of our perspective is the argument that measurement at the project level provides too narrow a focus for those who are responsible for multiple systems on an ongoing basis. Those managers want to know not only whether a system is right for its task, but also whether it is right for the other systems, and the future systems, of the organization. Their questions include:

* Does the application "fit" the overall architecture of the firm's systems? Or is it different in ways that would explain variances in development performance and subsequent contributions to firm-wide reusability?

* Does the application contribute a fair share to software reuse at the organizational level? Or does it fail to meet targets for software reuse? If so, does it contribute to reuse in subsequent development efforts to a greater extent than other projects?

* How large a portion of the software asset does a given system represent? How large a portion of the software liability did the system create? How do the two relate to each other?

It is no failing of project level metrics that they do not answer questions about software assets; they were never meant to answer them. Such questions become meaningful in the context of multiple systems.

2.2. Towards Software Asset Management: Recent Industry Experience

A number of organizations have begun to implement measurement programs directed at the management of the software asset. They are attempting to assess the leverage on development costs created by reusable software, the growth and evolution of repository-based software, and the extent and distribution of software assets as they exist at the enterprise level.

For example, Apte, Sankur, Thakur and Turner (1990) reported on the effects of

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

the successful implementation of a firm-wide software reusability strategy at Mellon Bank, a large regional commercial bank. This work was undertaken by a firm that already had significant experience with implementing programs to gauge the productivity of its traditional development and maintenance activities.

The First Boston Corporation, a large New York City-based investment banking firm, has examined its CASE-based development activities with an emphasis on the cross-application business value of software reusability. Banker and Kauffman (1991b) reported on order-of-magnitude productivity gains attributable to the firm's CASE technology and its software reusability strategy in a multi-year systems development effort that led to wholesale replacement of the firm's core trades processing systems. In this research, the authors found that it was necessary to extend current evaluative models in the software engineering economics literature to capture the effects of software reusability as a cost driver in CASE-based software development. An important ancillary finding of the research was that the resulting functionality of the software was perhaps even more important to the bank in sustaining its competitive position in a rapidly changing world of global electronic trading. (Clemons (1991) reinforces this point by indicating that an inability to rapidly deliver software in support of new business opportunities poses a "functionality risk" for software asset development.)

Subsequently, the senior management of the firm's software development organization moved to implement a program to measure the extent to which reuse was occurring at the project and the repository levels. The firm's CASE software has been enhanced to automatically compute function points, reuse levels, and other useful metrics, at both levels.

In each case, manual measurement of the software metrics that were collected would have been a difficult, time-consuming and expensive task. Yet without this information, management would be hard pressed to monitor the extent to which reuse creates leverage in software development. Fortunately, object- and repository-based

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

CASE does much to change that.

3. SOFTWARE REUSE: AN APPLICATION OF REPOSITORY EVALUATION

Some of the most interesting questions for firms that have invested in CASE technology center on the issue of software reusability; this is an area in which CASE-based development has the potential to create extraordinary value (Bouldin, 1989; McNurlin, 1989; Moad, 1990; Pollack, 1990; Sentry, 1990). However, this promise has not been broadly substantiated in industry, since the technology has only recently been deployed, nor in software development performance research, which has only recently begun to examine CASE-based development platforms (Kemerer, 1989; Norman and Nunamaker, 1989; Nunamaker and Chen, 1989; Scacchi and Kintala, 1989; Senn and Wynekoop, 1990).[1]

The success of a strategy that emphasizes software reusability can only partly be evaluated at the level of the software development project. Project-level statistics can tell us how successful a project is at reusing existing code, but cannot differentiate between projects whose new code is reused by subsequent projects and those whose code is destined for only a single use. The former software represents a corporate asset in a sense that the latter does not. A program of repository-level monitoring can identify, and help to control, such projects. Of potentially greater importance is the ability of such measurement to answer enterprise-level questions. For example:

* Is most of the observed reuse limited to a small proportion of the software? What are the observed characteristics of reused software? Can reuse levels be increased by imparting these characteristics to software intended for reuse?

---

[1]The reader interested in obtaining additional background on software reuse would benefit from looking at three recent papers which bring the literature up to date: Karimi (1990), Kim and Stohr (1991), and Banker, Kauffman, Wright and Zweig (1990). For an older, but still useful examination of the state-of-the-art in software reusability, see the *Special Issue on Software Reusability* of the *IEEE Transactions on Software Engineering*, September 1984. This issue contains articles with an overview of the statistics available at that time on reuse and then-current technical strategies to promote reuse.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

* What is the CASE technology's contribution to reuse? How much depends upon the way the technology is managed?

* Do expert programmers exhibit much higher levels of reuse than novices? Is reuse a skill that can be taught?

* How effective are efforts to design for reuse? Do objects specifically designed to be reusable show significantly higher levels of reuse?

These questions presuppose the existence of robust and readily implemented metrics that provide the right information. (Researchers have just begun to understand how to measure and interpret metrics for software reusability (Banker, Kauffman and Zweig, 1990; Gaffney and Durek, 1989).) This is not to say that organization-wide software reusability monitoring cannot be carried out in a traditional software environment. On the contrary, such monitoring is not only possible, but eminently worthwhile.

It is difficult, however -- sufficiently difficult that it is rarely done. And because the development environment does not support such measurement, when it is done, it tends to be a one-time occurrence, rather than part of an ongoing program. An integrated CASE tool, by maintaining a database of information about all the elements of a system in a single uniform repository, makes it practical to monitor the system continuously.

3.1. An Integrated CASE Environment (ICE)

We next discuss the use of repository-level measurement to assess the attempts of two organizations to realize high levels of software reuse through the adoption of CASE technology. The CASE technology implemented at both research sites was ICE (an acronym for Integrated CASE Environment -- not its actual name), which was deliberately designed with reusability as an objective.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

ICE is an integrated CASE environment of object-based design. Its objects include Screen Definitions, Report Definitions, Files, Data Domains, Fields, and Database Views, each class having its own procedures and semantics. Most of the procedural functionality of this language is embodied in high-level Rule Sets. Rule Sets are written in a fourth-generation programming language from which code is generated automatically and later compiled for the target machine. All interactions between objects are mediated by Database Views: If a Rule Set invokes a Screen Definition, for example, it will typically use one output View to send data to the terminal and one input View to receive data from the terminal. A Rule Set may also call an existing 3GL module. (It should be noted that ICE's objects are objects of the CASE environment rather than objects of the application environment. The 4GL is not an object-oriented programming language, though ICE can, and does, support object-based design.)

All the objects of the application systems are stored in a single repository. All calling relationships between objects are also maintained in this repository, in the form of entries to database tables. An overview of the contents of the repository of the research site is given in Table 1. Site One is the larger of the two, but Site Two's applications are more data-intensive, as may be inferred from the relatively large number of data objects. Neither site has created many 3GL modules for its new systems, but Site One is making use of a large inventory of 3GL modules (primarily modules which carry out highly specialized computations) from older systems which are being replaced.

Insert Table 1 about here

ICE implements software reuse by adding a calling relationship between a new object and one that is already in the repository. Beyond the obvious role this capability plays in facilitating reuse, it also makes it practical to monitor reuse, without having to examine individual programs, by analyzing the repository's database of calling

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

relationships.

3.2. Measurement of Software Reuse

The structure of the repository makes it practical to automate reuse analysis. An application system consists of a high-level Rule Set, designated as the root of that system, all the objects (mostly other Rule Sets) which it calls, and all the objects which *they* call, directly or indirectly. Collectively, these objects are structured as a hierarchy that defines the application. Since all these calls are stored in the repository in a uniform manner, an automated analyzer can use this data directly to determine which objects are called by, and thus belong to, which systems, and how many times each object is called.

The analyzer is used by designating a repository object as the root of a query. Depending on where this object resides in the hierarchy, the output could be an analysis of a subsystem, a system, or even the entire repository. The analyzer uses the information in the repository to navigate the calling hierarchy and identify all the objects called, directly or indirectly, by the root object, and to determine how many times each object is called.

Since the repository contains complete histories of each object (as well as the objects themselves) the analyzer has access to descriptive information about the objects, such as their age, the identities of their developers, and the systems for which they were originally created. (Note that it is imprecise to speak of an object as 'belonging' to any one application system. An object is part of any system which calls it.)

A number of measures of software reuse may be computed, depending on the purpose of the query. For the discussion that follows, reuse will be measured in terms of *reuse percentage*, which we define as the proportion of objects that were taken off the shelf, rather than programmed from scratch (Banker, Kauffman, Wright, Zweig, 1990; Gaffney and Durek, 1989). That is,

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

$$REUSE\text{-}PERCENTAGE = 100 * (1 - \frac{NUMBER\text{-}OF\text{-}UNIQUE\text{-}OBJECTS}{NUMBER\text{-}OF\text{-}OBJECT\text{-}CALLS})$$

In the absence of reuse, each unique object would be called only once, and the reuse percentage would be 0. In the example portrayed in Figure 1, there are four unique objects: A, B, C and D. But there are five object calls (counting the original invocation of A), as B and C each call D. In the absence of reuse, D would have to be replaced with two unique objects D1 and D2. Given Rule Set A as the root of this query, the analyzer would identify B and C as the objects called by A, and would identify D as being called by B and by C. This subsystem, then, has five calls for four unique objects: Reuse percentage is 100 * (1 - 4/5), or 20%.

Insert Figure 1 about here

A further distinction may be made between internal reuse and external reuse. *Internal reuse* is the multiple use of an object (or subroutine, or procedure, or module) within the project or application system for which it was originally written. *External reuse*, the use of an object originally written for another system, is more difficult to achieve, since it requires compatibility (planned or accidental) of design (Allen, Krutz and Olivier, 1990; Cohen, 1990) but it is, by that token, the source of the greatest potential gains. External reuse makes it possible to take advantage of the design efforts of previously developed software assets.

By measuring reuse over time, we can assess the success of the research sites in implementing a software reuse strategy through the adoption of ICE. We can also begin to open the black box of software reuse and what factors -- technological and otherwise -- determine the success of the reuse effort.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

Figure 2 presents a simple model of reuse that might motivate the design of a CASE-based tool to support reuse. The chance of reusing an existing object rather than writing a new one is seen to depend upon the availability of potentially reusable software, and upon the programmer's ability to find it. ICE supports reuse by maintaining a growing pool of reuse candidates within a single repository, by providing a keyword search mechanism for locating appropriate objects, and by automating the mechanics of reuse.

Insert Figure 2 about here

This view of the reuse process suggests a number of predictions:

1. *The pool of reusable objects will increase over time with the size of the repository, and so, therefore, will the level of reuse.*

2a. *Object belonging to the system currently being programmed are more likely to be known to the programmer, so there will be a high level of internal reuse.*

2b. *By a similar token, we expect programmers to exhibit high levels of reuse of objects that they wrote themselves. Both these familiarity effects may be mitigated by the presence of a good search mechanism.*

3. *Given a high level of reuse of familiar objects, we may expect reuse levels to be higher for larger systems, since they represent a larger pool of reusable objects and reuse opportunities.*

4. *Programmers with more experience at the site will be familiar with more of the software, and will therefore experience higher levels of reuse.*

3.3. Repository Growth and Software Reuse

Reuse levels were tracked at both research sites over the first two years of

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

application systems development.[2] Figure 3 presents the growth in Rule Set population and reuse during that time.[3]

Insert Figure 3a and 3b about here

It is immediately clear that our first prediction was incorrect: The repository grew steadily during this period. So did the experience of the programmers, since this was their first experience with ICE. Reuse percentage, however, achieved a strong initial value and never bettered it. The level of reuse did not grow as the pool of reuse candidates grew. Our second prediction, however, which was based on the belief that familiar objects were more likely to be reused, was borne out more strongly than expected -- sufficiently strongly, in fact, to offer an explanation for the failure of our first prediction.

3.4. Reuse of Familiar Objects

We predicted that programmers would be most likely to reuse objects from the system upon which they were currently working, as those would be the most easily identified as being appropriate for the task on hand. We also predicted, on the basis of the belief that familiarity was an important reuse factor, that programmers would exhibit a strong propensity to reuse software written by themselves. What we did not expect was the degree to which this would be true.

---

[2]The two sites had very different startup experiences. The data presented here is for the twenty-month period following the first development successes.

[3]Rule Sets are the 'backbone' of ICE application systems. They are also the most time-consuming objects to write. (3GL Modules could be more time-consuming, except that they are typically used in cases where special-purpose routines are already "on the shelf.") For these reasons, we concentrate upon Rule Sets for this analysis of software reusability.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

Figure 4a shows the relationship between internal and external reuse: 85% of all observed instances of reuse were internal. That is, if use was made of a previously written rule, that rule was almost always one that had been written for the same system.

Insert Figures 4a and 4b about here

This offers a potential explanation of the levelling off of reuse over time. Reuse appears to be driven by the pool of *familiar* code, rather than by the entire pool of reuse candidates. Each project is a self-contained universe (we assume that programmers will be most familiar with the code with which they are currently working than with that upon which other programming teams are working) and new projects derive little benefit from previous projects.

The propensity to reuse objects from the same application system could have a second explanation, as well: In the absence of an organization-wide effort to design for cross-application reuse, a system's own objects might naturally exhibit a better 'fit'. Our examination of the propensity of programmers to reuse their own software suggests that familiarity is indeed the driving force behind our observation.

Figures 4b shows the prevalence of self-reuse. Despite the presence of over 250 other programmers at Site One and over 100 other programmers at Site Two, over 60% of the reuse consisted of programmers reusing their own software.

If reuse is driven by the availability of familiar objects, we would expect to find, as we also predicted, that larger projects exhibit higher levels of reuse -- since they provide larger pools of familiar reuse candidates. This prediction was moderately supported. Figure 5 shows the relationship between system size and reuse. The correlation between these two factors was 37% ($p=0.09$) for twenty-two application systems at Site One and

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

58% (p=0.04) for thirteen application systems at Site Two.[4]

Insert Figure 5 about here

3.5. Individual Programmer Differences

As with so many software-related activities, a small number of outstanding programmers appear to account for a disproportionate amount of the reuse achieved. Figure 6 shows the distribution of programmer productivity and reuse. The top 5% of the programmers accounted for over 20% of the code and for over 50% of the reuse, with the top reusers achieving average reuse percentages as high as 75%.

Insert Figure 6 about here

It appears that reuse is learned: Reuse levels were consistently higher for programmers with larger total outputs. The correlation between these factors is 50% (p=0.03 for n=19) at Site One and 60% (p=0.0001 for n=76) at Site Two.[5] We considered the possibility that we were observing an attitude change over time, rather than the learning of a skill, with the high-reuse programmers simply being the ones who had been using ICE the longest, and had absorbed the reuse 'message'. The data did not

[4]Figure 5 uses a logarithmic scale to display system size, because order-of-magnitude differences between systems make a linear display difficult to interpret. In fact, though, the correlations between reuse and the *log* of system size at the two sites is exactly the same as that between reuse and system size: 37% and 58%, respectively.

[5]Of the 110 programmers at Site Two, only the 76 who wrote at least one Rule Set were included in this analysis. Our data for Site One represents a sample of 19 programmers out of 250.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

bear this out: The partial correlations, controlling for months of ICE experience, were within 1% of the raw correlations.

In summary, it appears that ICE provides capabilities which allow programmers to achieve high levels of reuse. Further, achieving reuse appears to be a skill which can be learned over time. However, the pattern of reuse suggests that there remains considerable unexploited reuse potential. Programmers are writing new objects rather than searching for reuse opportunities.

4. REPOSITORY EVALUATION: FACTORS AFFECTING SOFTWARE REUSABILITY

We interviewed developers to learn about the practice of software reuse from the perspective of the users of the CASE tools. These interviews revealed some technical barriers to the realization of code reuse opportunities. More serious, however, were the organizational barriers and disincentives to reusing software.

4.1. Search for Reusable Software

ICE makes the invocation of a previously written object trivial. All objects reside in the same repository, and are available for reuse. The main formal mechanism for identifying such an object, however, is a keyword search mechanism, the use of which often turns out to require more effort than programmers are willing to expend. (We found no indication that developers are failing to enter keywords into the index. It appears to be the case, though, that such keywords do not provide an efficient search mechanism. Given the relative ease of writing any single object, programmers are often reluctant to bother with an extended search.) This accords with the observations of other researchers (Banker and Kauffman, 1991; Palmer, 1991; Vassiliou, 1991) who suggest that search costs are one of the main barriers to software reuse.

4.2. Organizational Incentives

The more serious problem we identified revolves around incentives. The incentive

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

for programmers to reuse code is moderately weak. Reuse is viewed in a positive light at these sites, but it is not rewarded. There is little managerial monitoring of reuse levels, and programmers are valued -- as is usually the case -- for their ability to meet deadlines, rather than for their ability to match technical benchmarks. On the other hand, there are strong informal incentives for a programmer to *prevent* others from reusing his or her code.

The creator of an object is its 'owner,' and every reuse of that object is a potential call upon that owner to maintain the object in case of trouble -- most likely trouble arising from its use within an application for which it was not originally tuned and tested. Every reuse is also a constraint on the owner's subsequent ability to modify that object, since any modification must meet the requirements of all users of the object. (Incentives were not in place to motivate programmers to make their objects as reusable as possible in the first place. A strong change-control mechanism that could protect programmers was also lacking.)

In practice, programmers who wish to use an object from another application are strongly encouraged (by the other programmers, not by management) to copy the object in question, to rename it, and to use it as though it were a new object. We refer to this practice as "hidden reuse," a form of reuse which is not captured by the monitoring mechanism. (The related practice of "templating" is the dominant form of reuse in traditional application environments.) Hidden reuse achieves only some of the goals of software reuse: Coding effort and unit testing are reduced, but subsequent life cycle savings, particularly in maintenance, are not realized.

4.3. Preliminary Conclusions about Reusable Software

The initial drive for reuse at the research sites was premised upon the assumption that the primary determinants of reuse were technical -- that reuse could be achieved to the extent that we had a large pool of reusable objects, and that we had good tools for locating and using them. These expectations were correct, as far as they went, but they

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

did not go far enough. In particular, they did not consider the organizational prerequisites for successful reuse. The exceptionally high levels of internal reuse further suggested that design compatibility might have a strong effect upon reusability.

Figure 7 presents a revised model of software reuse, in light of the repository evaluation results presented in Section 3. The mostly technical factors which the earlier model presented as drivers of software reuse are still in place. And we note that the research sites did achieve strong initial levels of software reuse, with reuse percentages of about 35% at both sites, with the aid of the technical support provided by ICE. At this point, however, reuse appears to have reached a plateau.

Insert Figure 7 about here

The immediate barriers to higher reuse levels appear to be organizational. Reuse is encouraged, but it is not rewarded. What we have observed here is essentially unmanaged software reuse. Software reuse is encouraged to happen, but no organized effort is underway to make sure that it does happen. Programmers are not trained in reuse, programmers are not rewarded for reuse, and effort is made to monitor and manage reuse levels.

The weakest technical aspect of ICE with respect to software reuse is the keyword search mechanism, which appears to be unequal to its task. This weakness may also be related to the lack of architectural support for reuse. Although programmers are encouraged to take advantage of reuse opportunities as they arise, the various systems are not specifically designed for reuse; when reuse opportunities do arise, it is by

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

chance.[6]

The findings reflected in our repository evaluation and in our model suggest that integrated CASE technology can indeed contribute to high levels of software reuse, but that their full benefits can only be realized when corresponding changes are made in the way software development is planned and managed.

5. CONCLUSION

CASE technology promises to support the realization of the asset value of software in the most direct way, by supporting its reuse in subsequent development efforts. By definition, the fulfillment of this promise can only be evaluated by monitoring the organization's entire software asset. The repository-based design of integrated CASE systems makes it practical to automate such analysis. While such automation is not necessary in principle, an organization without it is unlikely to spare the resources to monitor its software assets on an ongoing basis. Management resources will be devoted to the more urgent priority of monitoring current projects.

This paper presents a new measurement paradigm, repository evaluation, that takes advantage of the facilities of the CASE environment in order to assess the extent to which a software reusability strategy is supported by that environment. The paradigm is illustrated through an analysis of reuse at two sites which are pursuing reuse by means of the same CASE tool.

Repository evaluation allowed us to critique a simple model of software reuse, and to suggest a richer one. It gave us preliminary answers to our original questions, and suggested a more focused set of questions whose answers we are in the process of pursuing:

---

[6]The kind of domain analysis that such design would require would also be likely to result in a vocabulary of keywords which would make the search mechanism more effective.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

* We asked whether a small proportion of the software accounted for most of the reuse, and found that it did: Approximately 15% of all Rule Sets are reused at least once. 15% of *those*, or under 3% of the total, account for half of the observed instances of reuse. We are now investigating the characteristics of heavily reused objects, to determine whether they can be used in the development of design guidelines.

* We investigated the extent to which the CASE technology supported reuse, and found that it enabled both sites to achieve steady-state reuse percentages of approximately 35%, but that higher levels probably depended on non-technical factors. We are now attempting to estimate the degree of unexploited reuse potential, and the costs of achieving it.

* We asked whether expert programmers were also better at reuse, and found that the highest levels of reuse were achieved by the most productive programmers. We have seen evidence that reuse is, at least in part, a learned skill. We are investigating the question of whether it is one that can be taught.

Such study can help us to further refine our understanding of software reuse. For example, in our presentation we have treated all objects as being equal in potential reusability and in value. In fact, the reuse of complex objects may yield far greater gains than that of easily-constructed objects. The same repository capabilities that allowed us to automate the analysis of reuse also enable us to automate a more detailed analysis of the objects of reuse, and to answer questions such as:

* What proportion of the *functionality* of a system and of its development cost does the reuse represent? Monitoring this offers the promise of allowing an organization to directly estimate the value of its reuse activities.)

* A high degree of software complexity in existing software makes it significantly more expensive to adapt (Banker, Datar, Kemerer and Zweig, 1991). Does it have a similar effect in a 4GL/CASE setting?

* Most systems have high levels of software redundancy, with overlooked reuse opportunities adding to ongoing development and maintenance costs. How can we measure this?

Function points, software complexity and redundancy are among the metrics which can

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

be automated in integrated CASE environments (Banker, Kauffman, Wright and Zweig, 1991), and which can provide additional repository evaluation tools.

The tools that enable us to address these questions can also aid managers in examining a broad range of software asset management issues which some organizations are just beginning to address (Miller, 1990) such as:

* The balance between the distribution of functionality in an organization, and the distribution of software investment dollars and of the software maintenance burden, measured in dollars per function point.
* The relative amounts of software functionality required to support various business areas.
* The relative productivity rates for continuing maintenance of aging systems.

The measurement of software assets at the repository level, and the potential for automating that measurement, makes such questions practical ones for managers to pursue.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

REFERENCES

Allen, K., Krutz, W., and Olivier, D. "Software Reuse: Mining Refining, and Designing," in *TRI-Ada '90 Proceedings*, December, 1990, pp. 222-226.

Apte, U., Sankar, C. S., Thakur, M., and Turner, J. "Reusability Strategy for Development of Information Systems: Implementation Experience of a Bank," *MIS Quarterly*, December 1990, pp 421-431.

Bailey, J. W., and Basili, V. R. "A Meta-model for Software Development Resource Expenditures," in *Proceedings of the 5th International Conference on Software Engineering*, 1981, pp. 107-116.

Banker, R. D., Datar, S., Kemerer, C. F., and Zweig, D. "Software Complexity and Software Maintenance Costs," Working paper #208, Center for Information Systems Research, Sloan School of Management, MIT, 1991.

Banker, R. D., and Kauffman, R. J. "Reuse and Functionality: An Empirical Assessment of Integrated Computer Aided Software Engineering (CASE) Technology at the First Boston Corporation." *MIS Quarterly,* Fall 1991.

Banker, R. D., and Kauffman, R. J. "Automated Software Metrics, Repository Evaluation and the Software Asset Management Perspective," Working Paper, Center for Information Systems, Stern School of Business, New York University, 1991.

Banker, R. D., Kauffman, R. J., Wright, C. and Zweig, D. "Automating Software Metrics for Repository-Based Integrated Computer Aided Software Engineering (ICASE) Performance Evaluation," Working Paper, Center for Research on Information Systems, Stern School of Business, New York University, 1990.

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

Barnes, H. B, and Bollinger, T. "Making Software Reuse Cost Effective," *IEEE Software*, 8:1, January 1991.

Basili, V. "Viewing Maintenance as Reuse-Oriented Software Development," *IEEE Software*, 7:1, January 1990, pp. 19-25.

Boehm, B., and Papaccio, P. N. "Understanding and Controlling Software Costs," IEEE Transactions on Software Engineering, 14:10, October 1988, pp. 1462-1477.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

Booch, G., "What is and What Isn't Object-Oriented Design." *Ed Yourdon's Software Journal*, 2:7-8, pp. 14-21, Summer 1989.

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

Chen, M., and Sibley, E. H. "Using a CASE-Based Repository for Systems Integration," in *Proceedings of the 1991 Hawaii International Conference on Systems Sciences*, Hawaii, IEEE, January 1991, pp. 578-587.

Clemons, E. "Evaluating Investments in Strategic Information Technologies," *Communications of the ACM*, January 1991.

Cohen, S., "Process and Products for Software Reuse in Ada," in *TRI-Ada '90 Proceedings*, December, 1990, pp. 227-239.

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.

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

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

Kemerer, C. F. "An Agenda For Research in the Managerial Evaluation of Computer-Aided Software Engineering (CASE) Tool Impacts," *Proceedings of the 22nd Hawaii International Conference on Systems Sciences*, Hawaii, IEEE, January 1989.

Kim, Y., and Stohr, E. A. "Software Reuse: Issues and Research Directions," *Proceedings of the 25th Hawaii International Conference on Systems Sciences*, Hawaii, IEEE, January 1992, forthcoming.

Lenz, M., Schmid, H. A., and Wolfe, P. F. "Software Reuse Through Building Blocks," *IEEE Software*, 4:4, July 1987, pp. 34-42.

McNurlin, B. "Building More Flexible Systems", *I/S Analyzer*, October 1989.

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

Miller, J.C. "Software Metrics at the General Electric Corporation", presented at Strategic Information Architecture Workshop, SEI Center, Wharton School of Business, University of Pennsylvania, June 22 1990.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

Moad, J. "The Software Revolution," *Datamation*, February 15, 1990, pp. 22-30.

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.

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

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

Palmer, C., "Software Reuse -- More than Just a Coding Issue," presented at the USAF/STSC - HQ USAF/SC Joint Software Conference, Salt Lake City, Utah, April 17, 1991.

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

Rombach, H. D. "Software Reuse: A Key to the Maintenance Problem," *Information and Software Technology*, 33:1, January/February 1991.

Scacchi, W. "Understanding Software Productivity: A Comparative Empirical Review," in *Proceedings of the 22nd Hawaii International Conference on system Sciences*, Hawaii, IEEE, January 1989, pp. 969-977.

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.

Sentry Market Research, *CASE Research Report*, Westborough, MA, 1990.

Vassiliou, Y., Personal communication, April, 1990.

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

Object Type	Number of Objects
Rule Sets	8892
Screens	7230
Domains	4200
Files	4236
3GL Modules	6062
Fields	6266
Views	6755

Table 1: An Overview of the ICE Repository

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

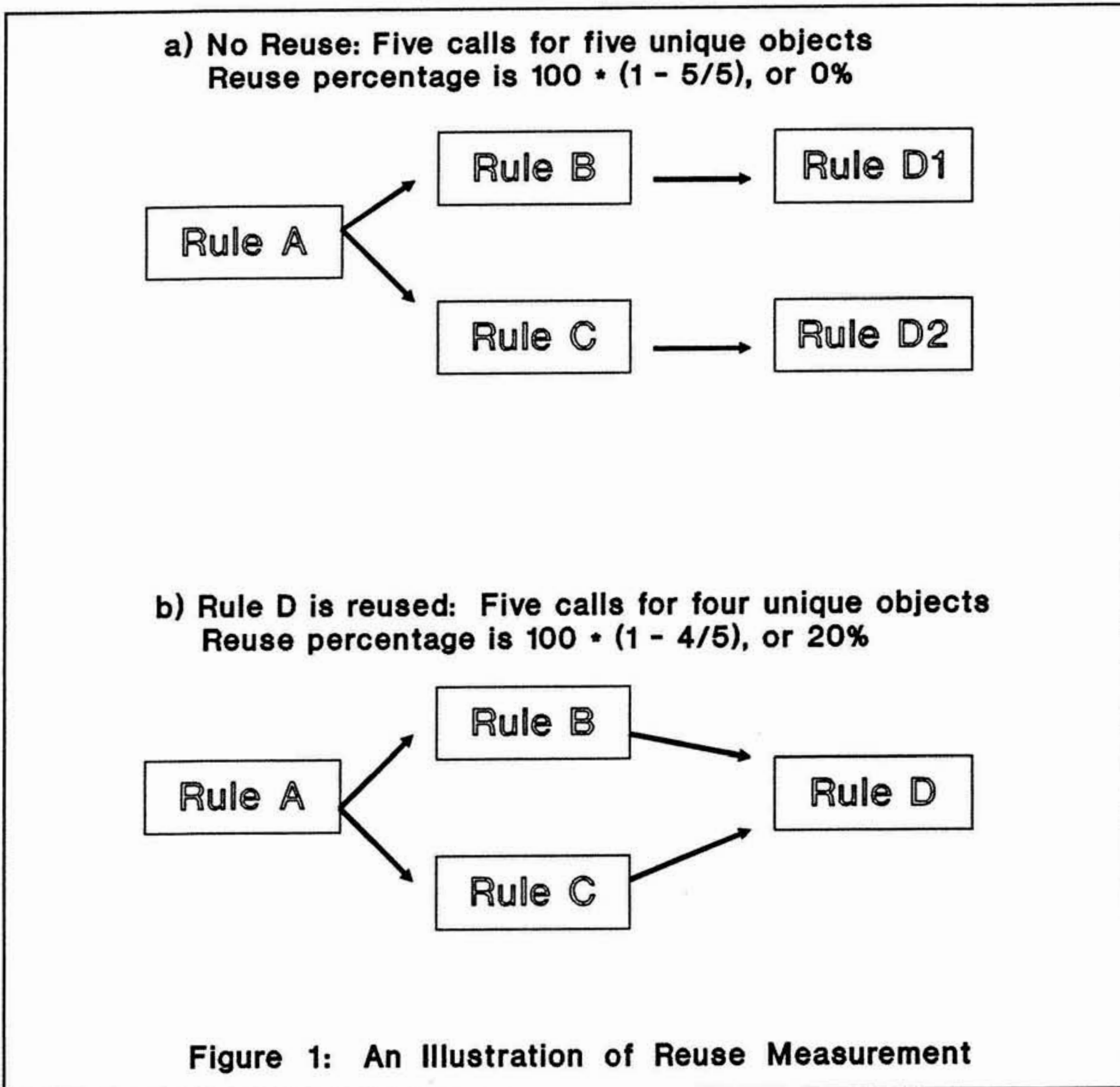


Figure 1: An Illustration of Reuse Measurement

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

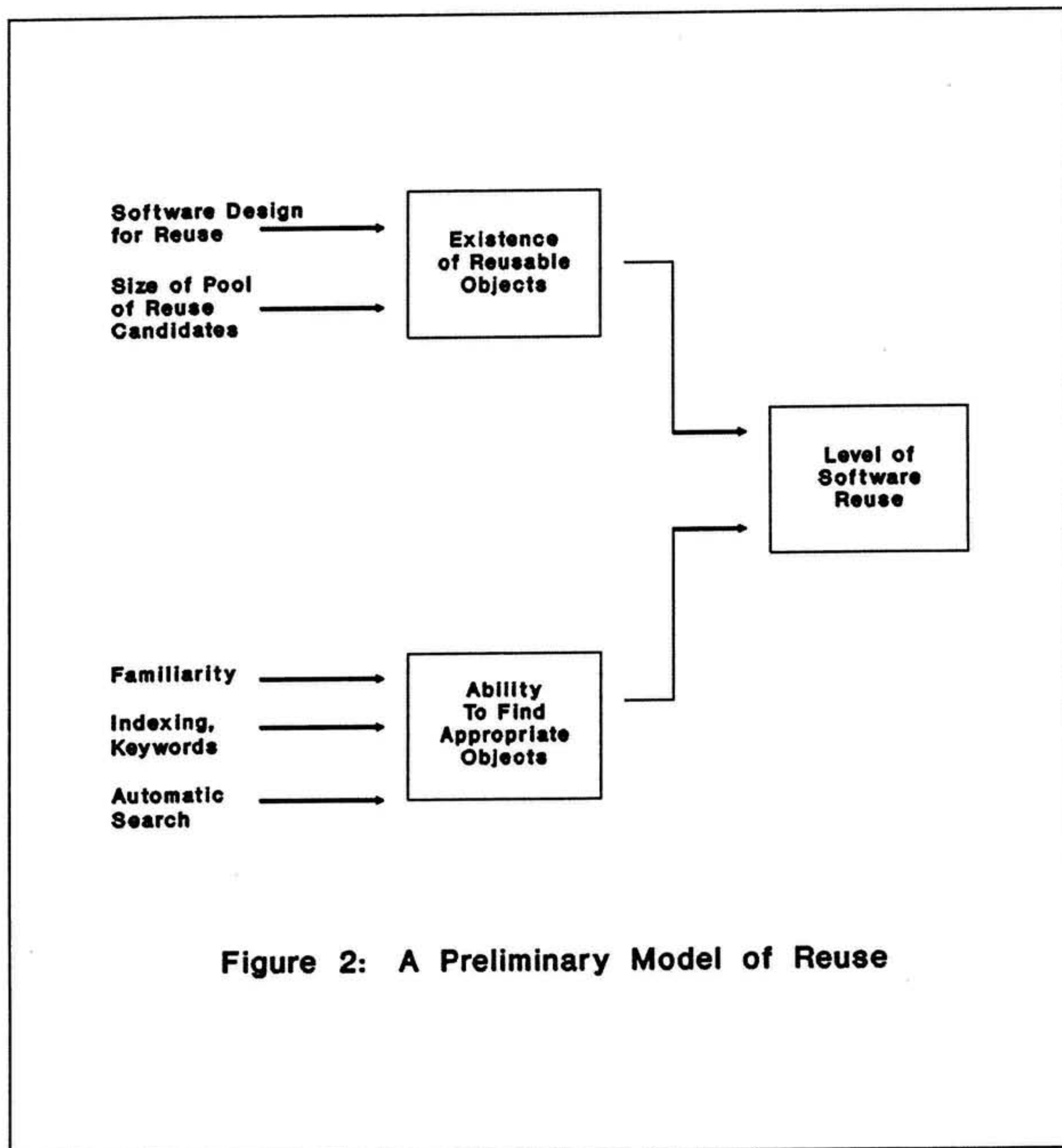


Figure 2: A Preliminary Model of Reuse

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

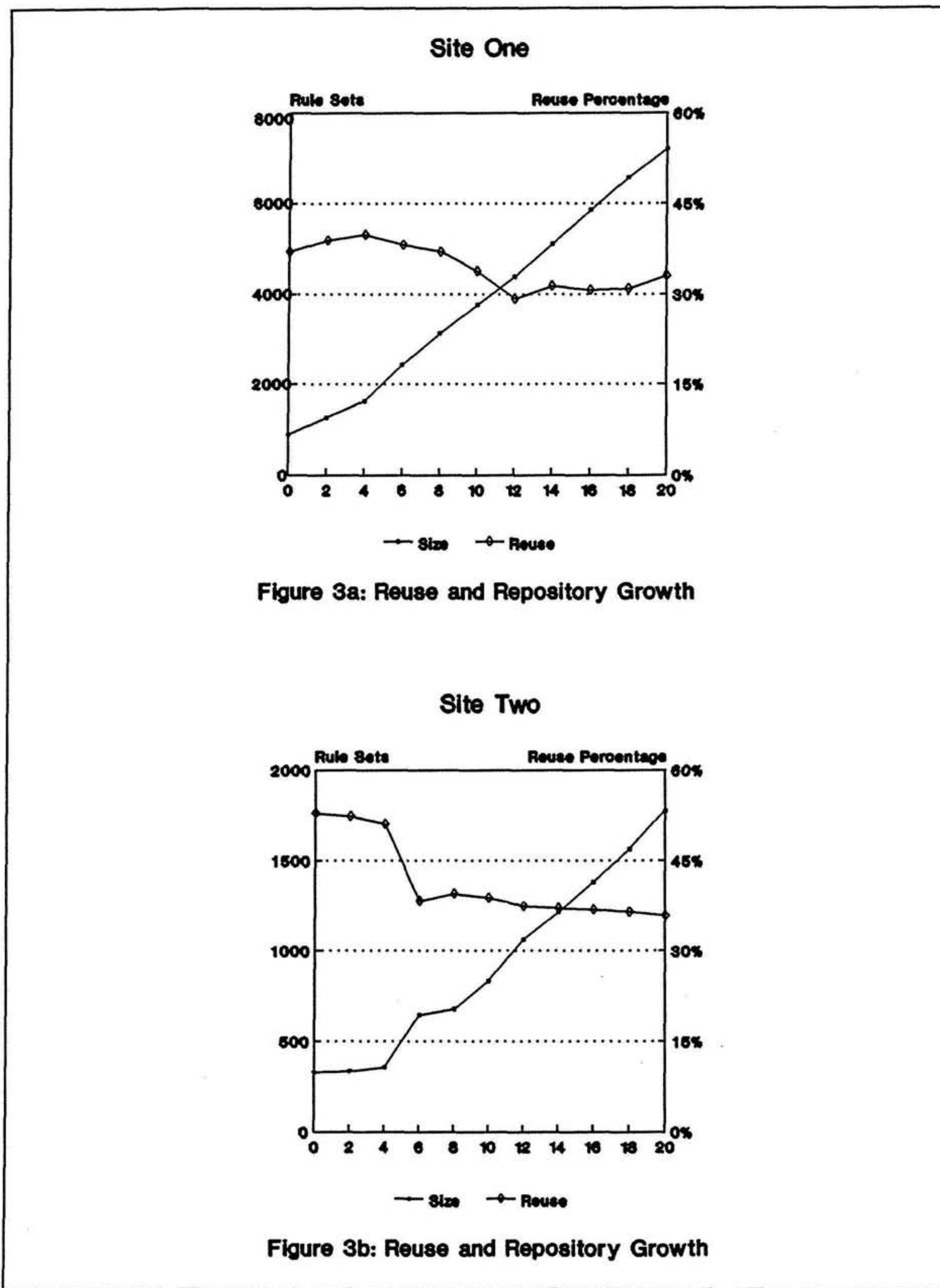


Figure 3a: Reuse and Repository Growth

Figure 3b: Reuse and Repository Growth

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

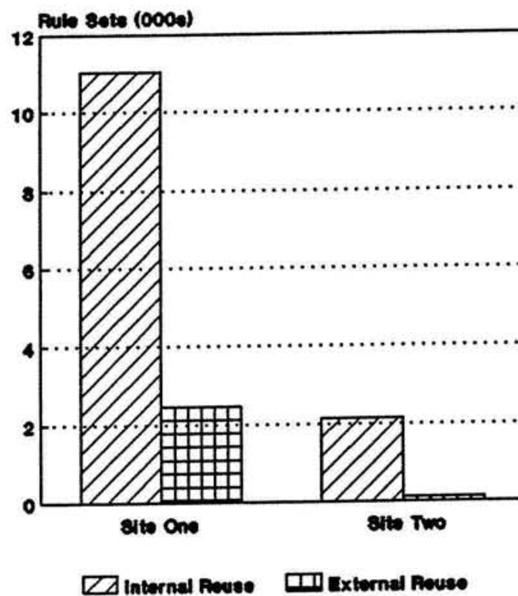


Figure 4a: Internal and External Reuse

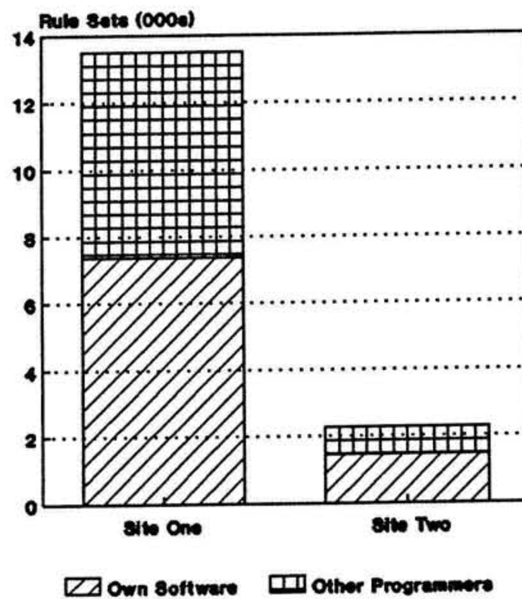


Figure 4b: Reuse of Own Software

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

Figure 5: Reuse and System Size

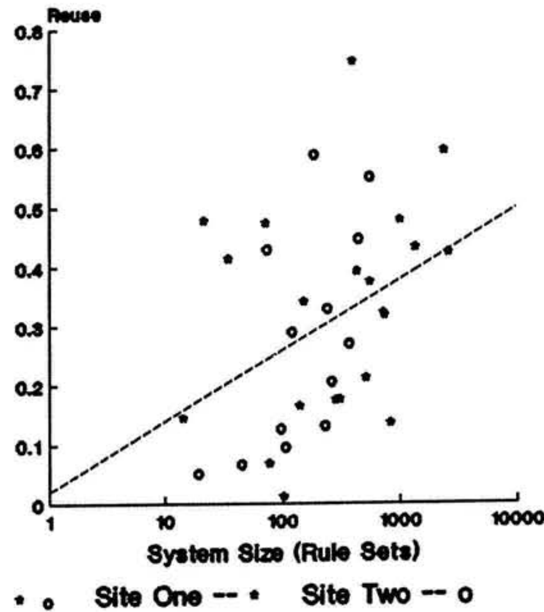
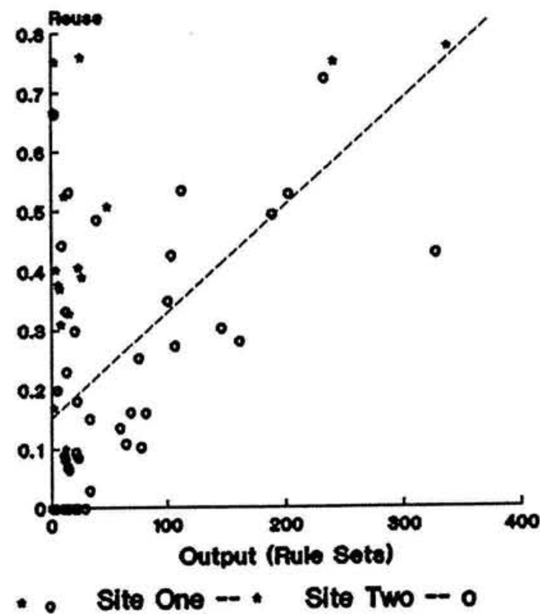


Figure 6: Reuse and Programmer Output



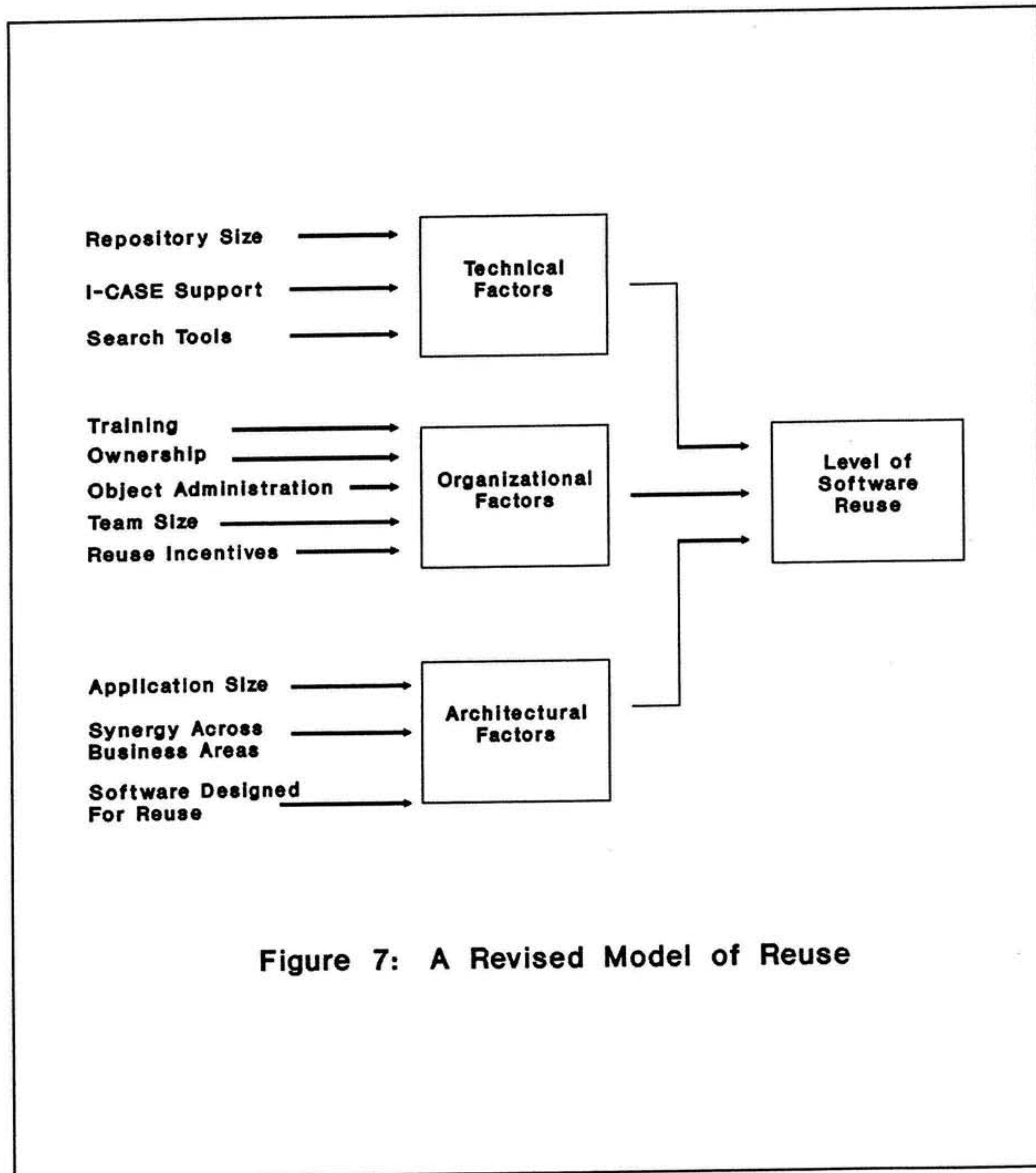


Figure 7: A Revised Model of Reuse

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32

Center for Digital Economy Research
Stern School of Business
Working Paper IS-91-32