1993

Repository Evaluation of Software Reuse: An Empirical Study

R. D. BANKER

Robert J. Kauffman
Singapore Management University, rkauffman@smu.edu.sg

D. Zweig

DOI: https://doi.org/10.1109/32.223805

Repository Evaluation of Software Reuse

Rajiv D. Banker, Robert J. Kauffman, and Dani Zweig

Abstract— **The traditional unit of analysis and control for software managers is the software project, and subsequently the resulting application system. Today, with the emerging capabilities of computer-aided software engineering (CASE) and corresponding changes in the development process, productivity gains can be realized by reusing portions of the organization's inventory of existing application designs and code. With this opportunity, however, comes the need to monitor software reuse at the corporate level, as well as at the level of the individual software development project. Integrated CASE environments can support such monitoring. We illustrate the use and benefits of repository evaluation of software reuse through an analysis of the evolving repositories of two large firms that recently 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.**

Index Terms— **CASE, computer-aided software engineering, domain analysis, organizational learning, repositories, software metrics, software reuse.**

I. Introduction

TRADITIONALLY, the management of software development has focused upon the individual software project. Managers are evaluated, in turn, on the basis of their projects' success in meeting cost and quality targets. Some organizations are devoting resources to process improvement, so that projects may be held to increasingly high standards, but even here, in all but the most mature organizations, the emphasis is on project-level monitoring [25]. Yet there is a range of insights that can only be attained through the monitoring and management of the software inventory at the level of the entire firm.

The example upon which this paper focuses is that of software reuse in an integrated computer-aided software engineering (CASE) environment built around an object repository. Software reuse, the incorporation of previously developed software elements into a system under development, has shown itself to yield substantial productivity benefits, even in traditional development environments.[1] CASE technology can provide considerable support for software reuse.

A number of industry observers have pointed to the special potential for development productivity and software quality improvements, when development occurs using CASE tools [8], [9], [16], [34]–[36], [40], [27]. The emergence of CASE tools that emphasize software reuse can mean that much of the real value of modular software will be derived from the extent to which it 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 opportunities for competitive advantage still exist in the business areas that the software is meant to support; and
- be leveraged across projects and areas of the firm in support of multiple businesses.

Meanwhile, recent empirical research has begun to uncover the extent of those gains [2]–[4].

The time-worn epithet that "you can't manage what you can't measure" clearly applies here. Reuse, by its nature, is an activity that spans multiple projects and application systems enterprisewide. To manage such reuse requires monitoring the firm's software at the level of the organization or enterprise. Even relatively simple metrics, collected at that level, can answer key questions for senior managers that traditional monitoring does not address.

Repository-based integrated CASE environments make the collection of such metrics practical. A repository maintains all of a corporation's software and, more importantly, all relevant information about that software, including its design, its history, and its interactions with other system elements. By analyzing software reuse at the repository level—what we call *repository evaluation*—we can cut across multiple projects to ask questions such as:

- What kinds of objects are most likely to be reused?
- Under what conditions is reuse most likely to occur?

This can lead, in turn, to a shift away from single or isolated software product-oriented questions to a new focus on more development process-oriented questions, such as:

Manuscript received October 23, 1991; revised December 9, 1992. This work was supported in part by the Nippon Electric Corporation and by the U.S.–Japan Business and Economics Research Center, Stern School of Business, New York University. Recommended by R. Selby and K. Torii.

R. D. Banker is with the Department of Accounting and Information Systems, Carlson School of Management, University of Minnesota, Minneapolis, MN 55455.

J. J. Kauffman is with the Department of Information Systems, Stern School of Business, New York University, New York, NY 10012.

D. Zweig is with the Department of Administrative Sciences, Naval Postgraduate School, Monterey, CA 93943.

IEEE Log Number 9207593.

[1] See, for example, Cavaliere's [15] report on the software reuse program at the Hartford Insurance Group, Lanergan and Grasso's [26] review of Raytheon's achievement of 50% productivity gains through software reuse and elimination of redundant software, and Cusamano's [18] discussion of efforts to reuse software among major Japanese electronics firms. For an overview of the key references in the software reuse literature, see the books by Biggerstaff and Perlis [11], [12], Freeman [22], and Tracz [46], and the articles published in two special issues of IEEE TRANSACTIONS ON SOFTWARE ENGINEERING (vol. SE-10, Sept. 1984) and *IEEE Software* (July 1987); Hooper and Chester [24] offer a useful update.

- Do technical advances in the development methods increase reuse to the same extent in different environments?
- Do differences in organizational structure lead to different levels of success in managing software reuse?
- What can be done to encourage more software reuse?

In this paper we will use automated repository evaluation to explore and interpret the experiences of two firms, the First Boston Corporation, a large New York City-based investment bank, and Carter Hawley Hale Information Services, the information systems organization of a large California-based retailing firm. A repository-based integrated CASE tool called High Productivity Systems (HPS) was deployed at both sites. Both firms believed that productivity increases in software development would only become possible through significant changes to their software development processes, and both firms considered software reuse to be a key element of the process improvement they sought. But, as the discussion will show, the firms took contrasting approaches to its tactical implementation.

II. Software Reuse and Reuse Measurement

Most of the attempts to implement formal programs of software reuse have been initiated within the past decade. Such programs rely heavily upon technological (CASE) support and high levels of process maturity.

A. Software Reuse

Extensive reuse in the construction phase has been shown to increase productivity by 20% or more [30], [3], [4], through the use and invocation of previously developed software modules. Greater productivity gains may be achieved by extending reuse to other phases of the software life cycle.

Reuse Throughout the Life Cycle: As modern software development practices increasingly emphasize phases of the life cycle other than programming, it becomes increasingly profitable to extend reuse efforts to those phases. Early in the life cycle, it is possible to reuse system architecture, and data structure and data model elements [19], [26], as well as the abstract representations of systems that are provided to the people who do the coding work [30]. When the opportunity arises, it may even be appropriate to reuse application prototypes and partial systems [37]. Later in the life cycle, it is possible to reuse existing code, particularly where prior development efforts have left behind well-documented code. For example, see the discussions of the Reusable Software Library (RSL) at Intermetrics Inc. in [13], and Westinghouse Electric's Reusability Search Expert (REUSE) in [33]. Even later, there is potential for the reuse of test routines and test data [43].

The benefits of reuse are enhanced when the software development methodology focuses on the reuse of entire modules [32] and software objects [31]. These may embody analysis and design efforts, as well as code, and prior testing and documentation, as well. When the activities involving reuse spread throughout the life cycle are linked by a methodology (for example, SSADM, information engineering or object-oriented design and construction) or an integrated tool set (as is the case with integrated CASE tools such as Texas Instruments' IEF, Andersen's FOUNDATION or Seer Technologies' HPS), software reuse offers the potential to create even greater long-term benefits [29], [41], [42].

Horizontal and Vertical Reuse, and Domain Analysis: The success of a program of software reuse depends upon the degree of commonality among the applications across which software is shared. Prior research distinguishes between reuse across vertical and horizontal domains [45]. *Vertical reuse* can occur when the majority of the applications built by software developers are representative of a single kind of data processing activity, and many software objects that are employed by one can be shared among the others. *Horizontal reuse* , by contrast, occurs across a broad range of application areas.

According to [24], horizontal reuse is more often employed and better understood than vertical reuse. Organizations that operate across different, highly technical domains, where little knowledge is readily transferred across businesses are likely to emphasize horizontal reuse. The software reuse programs undertaken by Raytheon [26], Hitachi [18], and the National Aeronautics and Space Administration [33], and the hypertext reuse search interface to unrestricted software at the Jet Propulsion Laboratory [10] are good examples.

Vertical reuse occurs less frequently. Such reuse offers greater potential benefits, but requires developers to first carry out a relatively thorough domain analysis, in order to design systems with the greatest possible commonalities. Prieto-Diaz [39] offers a useful introduction to domain analysis, and indicates that its use to date has tended to be ad hoc; the analysis process itself, in his view, is more an art than a science, and only with time can appropriate design decisions be made so as to optimize design for the purposes of reuse. Still there is a growing number of examples of vertical reuse. Examples that have been reported in the literature include the reusable software development program pursued by the Hartford Insurance Group [15] and McNicholl *et al.*'s [28] software reuse project in the domain of missile guidance systems.

We expect that vertical reuse will increase over time with the increasing sophistication of the CASE tools that support the functional and technical design activities. Although horizontal reuse is likely to offer more easily implemented reuse opportunities, vertical reuse offers higher payoffs, since it takes place across systems with higher degrees of potential commonality. In the absence of careful domain analysis, though, one expects vertical reuse to fall short of its potential.

Reuse Search, Adaptation, and Incorporation Costs: A major element that will determine the success of a software reuse program is the relative magnitude of two costs:

- the cumulative cost of locating, adapting and incorporating an appropriate existing software object or 3GL module into a new application, and
- the cost of building the same function from scratch and incorporating it into the new software, thereby eliminating the search and adaptation costs.

Search and adaptation can represent a significant cost to a well-meaning developer who is interested in reusing software [22]. The research suggests that search costs alone may often be too high, causing a developer to end a search prior to locat-

ing the appropriate reusable software. One response to these findings has been an effort to develop classification methods for potentially reusable software. (See, for example, [38] or [14].) A second has been the creation of tools and techniques to assist the developer in her search. The approaches include facet classification analysis [38], rule-based retrieval [21], and hypertext search [10], among others.

Gaffney and Durek [23] presented an economic model of software reuse that reflects the costs of porting, adapting, and incorporating reusable software. The authors argue that the value that reuse can deliver must be weighted by the costs that developers experience as they sort out these problems, relative to the total proportion of the application that results from reuse. Bullard *et al.* point to varying component quality when horizontal reuse occurs, and indicate that the major reuse costs come in verifying and validating their performance.

Templating, Mining, and Refining: It is common knowledge among software development researchers that there is widespread, often informal, application of partial reuse approaches, such as templating new functions from similar old ones, mining existing code to pull out just the relevant pieces and refining existing code to serve a different purpose [1]. The benefits of these techniques, however, are largely restricted to the coding phase of the software life cycle.

B. Measurement of Software Reuse

Software reuse is commonly measured as a ratio of reused code to the total amount of code in a given system. Such measures focus upon code to the exclusion of the products of other development phases, but they have the virtue of objectivity.

For example, Toshiba computes the percent of the lines of debugged and delivered *equivalent assembler source lines* (EASL) that were incorporated into an application from elsewhere with little or no modification [18]. The Software Productivity Metrics Working Group of the IEEE (1992) has proposed that reuse be measured as the number of source lines of code (SLOC) incorporated in a system without modification, divided by the total number of SLOC in the system. Note that these metrics consider only instances of reuse in which adaptation costs can be ignored.

We have been engaged in a program of research on productivity in object-based CASE environments. Our findings suggest that for such environments it is appropriate and meaningful to measure reuse in terms of entire objects, rather than SLOC [3]–[5]. In related work, [6], showed that "object counts" were found to be more useful than SLOC as a basis for software cost estimation at such sites (they yielded post hoc estimates that were as accurate, and they were available far earlier in the life cycle) and were far more meaningful to programmers and project managers. Objects have the added advantage that they embody analysis and design efforts as well as the product of the coding phase.

III. The Research Sites

In this research, reuse levels were tracked over the first two years of application systems development with the newly deployed CASE tool, HPS, as two research sites: The First Boston Corporation (FBC) and Carter Hawley Hale Information Services (CHH).

Both FBC and CHH exhibited strong managerial support for software reuse. Both sites believed themselves to have application systems with high degrees of commonality, and considerable scope for reuse, and programmers at both sites were encouraged to take advantage of HPS's support of reuse. Although the two sites followed different philosophies of reuse management, their experiences turned out to be remarkably similar.

A. The First Boston Corporation

This investment bank faced two key problems in the mid-1980's. It foresaw itself losing the ability to control its development costs and to produce the increasingly complex systems it needed in order to remain competitive. Efforts to engineer software costs were hamstrung by application complexity, which required expensive developer expertise, and the development of applications running cooperatively on multiple hardware platforms. Senior management believed that the bank would be unable to control the costs of software development five years into the future. At the same time, strategic analysis indicated that the bank's competitiveness depended upon its ability to bring software-based trading products to market ahead of, or at least in synch with, the competition.

To senior management's dismay, a 1986 survey determined that there would be no commercially available software development tools within five years which would support cost-effective expansion of the firm's systems. Without substantial changes in the firm's software development methodology, it was just a matter of time before the bank's systems would be unable to meet the demand for increased financial market trades processing in a 24-hour a day, global market. At this time, First Boston employed over 700 person-years of full-time-equivalent software labor (in-house or contracted), an expense that was growing more rapidly than any other cost category.

The firm's solution was to develop its own integrated CASE tool, and to emphasize software reuse. The bank began the development of HPS in 1987. When HPS was first deployed, software developers reported that it took about two to three months to travel about 70% of the way down the learning curve. In addition to learning how to work with the CASE tool set, developers and project managers reported that they were simultaneously learning how to reuse software in that environment. Part of that process involved learning the extent to which it was necessary to concentrate on application design, in lieu of technical design or construction. Most developers whom we interviewed reported that development under HPS encouraged the substitution of design labor for construction labor.

B. Carter Hawley Hale Information Services (CHH)

The complexity of the data processing requirements of multibusiness, multiunit retailing firms also grew dramatically during the 1980's. CHH's Information Services unit was under pressure to deliver a new generation of retailing applications

that would improve the flow of store and product performance data to senior managers, enabling them to improve inventory management and refine product pricing. These systems had to support extremely high transaction volumes, at acceptable costs, at a time when slowing economic growth and increasing competition were intensifying cost pressures in the retailing industry.

Beginning in mid-1989, CHH carried out what it called its "benchmark project," to determine the extent to which the application design and software construction philosophies embodied in HPS were workable for its own software development. With technical challenges akin to those of FBC, CHH investigated the extent to which HPS might enable the development of complex transaction processing and multilevel management reporting systems operating cooperatively across multiple platforms. In addition, management hoped to evaluate HPS in terms of its ability to support rapid prototyping of applications that later would be deployed to the buying and store organizations, where the usability of a system was of paramount concern.

In the process of evaluating the results of the "benchmark project," CHH's software development managers identified software reuse as a key to improved productivity. They came to believe that software products could be produced most efficiently using HPS if there were many opportunities to reuse software objects built for other projects. For this reason, and with the benefit of FBC's experiences, CHH chose to establish a project whose sole purpose was to produce software objects representing the core functions of its retailing domain, when it adopted HPS in late 1989.

IV. Software Reuse in HPS

HPS was designed to support the development of widely distributed application systems cooperatively processed on a range of platforms. Developers are shielded from the technical complexities entailed by such systems. They do not have to develop platform-dependent code, and the programming of communications between platforms is largely automated by what the developers call "middleware." The design of HPS emphasizes productivity improvement through object-based development, software reuse, and an integrated family of CASE tools.

A. HPS: An Integrated CASE Environment

HPS is an integrated CASE environment of object-based design. Its first applications were in the investment banking industry, where it had to support the development of trading systems, which required global distribution and 24 hour availability. Further, performance requirements demanded that these systems run cooperatively on several different platforms: High-function workstations programmed in C++ had to communicate with central DB2 databases residing on mainframes programmed in COBOL. Minicomputers programmed in PL/I linked the workstations and mainframes with each other and with the market, providing real-time communication and pricing information. The challenge was to create and maintain such systems without having to support and interface three sets of programmers, as had previously been the case.

```
map SCG_CUST_ID of SCG_CTRCT_BOX_LST_X to
   SCG_CUST_ID of SCG_CTRCT_BOX_SQL_FET_X
map SCG_FIRST_NM of SCG_CTRCT_BOX_LST_X to
   SCG_FIRST_NM of SCG_CTRCT_BOX_LST
map SCG_LAST_NM of SCG_CTRCT_BOX_LST_X to
   SCG_LAST_NM of SCG_CTRCT_BOX_LST
use rule SCG_CTRCT_BOX_SQL_FET
converse window SCG_CTRCT_BOX_LST
caseof WINDOW_RETCODE
case 'BOXLST.BOXFLD' 'OK'
        map 'SCG_CTRCT_BOX_FET_OCC' to
          VIEW_LONG_NAME of GET_SELECTED_FIELD_X
        use component GET_SELECTED_FIELD
        map SCG_CTRCT_.....
        .....
        return
endcase
```

Fig. 1. An HPS rule set.

HPS supports a number of predefined object types, including Screen Definitions, Report Definitions, Files, Data Domains, Fields, Database Views, and Rule Sets, each class having its own procedures and semantics. The Rule Sets are the backbone of an HPS application system. Most of the procedural logic of HPS applications is embodied in the Rule Sets (see Fig. 1 for an example), which are written in a fourth-generation programming language. Rule Sets are the most labor-intensive HPS objects to create, and our discussion of reuse in HPS will focus upon the reuse of Rule Sets.

Other object types have more specialized functions. For example, Screen Definitions are created by a screen-painting utility to define a window's format, input and output fields, and front-end data validation. Report Definitions are created by a report-generating utility to define a report's output field and format. 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.[2] For example, FBC was able to make considerable use of a library of optimized 3GL routines for specialized financial computations.

Third-generation code (PL/I, COBOL, or C++, depending on the designers' decisions as to which platforms would be most appropriate) is generated automatically from the HPS objects, and later compiled for the target machines.

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 DB2 database tables. All such relationships are of the form Object1-uses-Object2.[3]

Once an object has been created, it may be incorporated into an application system by adding a calling relationship between that object and one which is part of the target application system. Similarly, HPS implements software reuse by adding a calling relationship between a previously-created object and

[2] It should be noted that the HPS object types described here are objects of the CASE environment, rather than objects of the application environment. The 4GL is not an object-oriented programming language, though HPS can, and does, support object-based design. For more information about the design of HPS, see [7].

[3] To be more precise, "uses" is restricted to a Rule Set calling another Rule Set. A Rule Set calling a Screen Definition, for example, would have a different operator, and somewhat different semantics.

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 system documentation or program code, by analyzing the repository's database of calling relationships.

B. Reuse Measurement

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 which it calls, and all the objects which they call, etc. Collectively, these objects are structured as a hierarchy that defines the application. (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.)

Once we have identified the objects of an application system, the information in the repository allows us to identify the application system for which each object was originally created, and to count the number of times each object is called within the current system.

A number of measures of software reuse may be computed, depending on the purpose of the analysis. For the discussion that follows, reuse will be measured in terms of *REUSE PERCENTAGE*, which is defined as the proportion of object calls that represent the reuse of unmodified, previously created objects, rather than the initial creation of new objects:

$$REUSE\ PERCENTAGE = \left(1 - \frac{NUMBER\ OF\ NEW\ OBJECTS\ BUILT}{TOTAL\ NUMBER\ OF\ OBJECTS\ USED}\right) * 100$$

where

1) *NUMBER OF NEW OBJECTS BUILT = the number of new objects that had to be created from scratch for the application system and*
2) *TOTAL NUMBER OF OBJECTS USED = the number of objects the application system would contain in the absence of reuse, i.e., if a new objects had to be written for every cell.*

Note that objects that are reused multiple times are considered to represent multiple instances of reuse; this metric focuses on the total benefit attributable to reuse [3], [4]. Fig. 2 illustrates the measurement of software reuse.

In the example in Fig. 2, there are four unique objects: A, B, C, and D. But there are five object calls (counting the original invocation of A), since B and C both call D. This subsystem, then, has five calls for four unique objects: *Reuse_Percentage* is 100*(1 − 4/5), or 20%. In the absence of reuse, object D would be replaced by two unique objects, D1 and D2. The subsystem would have five object calls and five unique objects, for a Reuse_Percentage of 0%.

A further distinction may be made between internal reuse and external reuse. *Internal reuse* is the multiple use of an object (or, in other environments, a subroutine, procedure, or module) within the application system for which it was originally written. *External reuse*, the use of an object originally written for another application system,[4] is more difficult to achieve, since it requires compatibility (planned or accidental) of design [1], [17]. HPS programmers need not distinguish between the two forms of reuse, but the distinction may be important to the management of reuse. Some organizations only reward external reuse.

[4]External reuse can be vertical or horizontal, depending on whether or not the systems belong to a common domain.

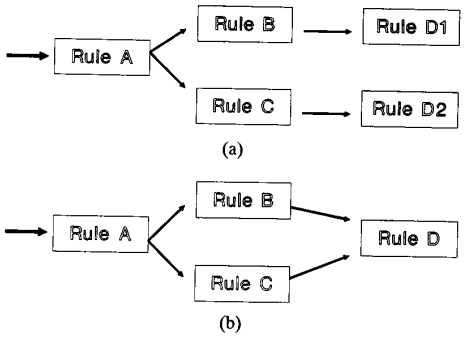


Fig. 2. An illustration of reuse measurement. (a) No reuse: five calls for five unique objects. Reuse percentage is $100 * (1 - 5/5)$ or 0%. (b) Rule D is reused: five calls for four unique objects. Reuse percentage is $100 * (1 - 4/5)$ for 20%.

C. Repository Evaluation in HPS

HPS stores all the objects of all its application systems, the calling relationships linking those objects, and a considerable amount of information *about* the objects, in the same easily-accessible repository, an architecture that makes it highly practical to automate repository evaluation. A suite of database access routines has been created to monitor and analyze the repository: we can determine when each object was created, by whom, and for which application system. We can identify the objects that call any given object, the objects it calls, and the application systems in which it is used. We can also analyze individual objects in greater detail, determining, for example, what data is passed between any pair of objects. This has made it possible to develop automated function point and software reuse analyzers.

By analyzing the entire repository over time, we can assess the success of the research sites in implementing a software reuse strategy through the adoption of HPS. We can also begin to open the black box of software reuse and identify the factors—technological and otherwise—that determine the success of the reuse effort.

V. Reuse Predictions and Observations

Our earlier discussion of the software reuse literature reflected the primarily technical focus of the research in this field: application domains are more or less conducive to reuse, cataloging schemes are more or less successful in guiding the search for reuse opportunities, and reuse is constrained by search and adaptation costs. The initial reuse efforts at FBC and CHH reflected a similar technical focus.

A. Simple Model of Reuse

Fig. 3 presents a simple model of reuse. In this model, the chance that a programmer will reuse an existing object,

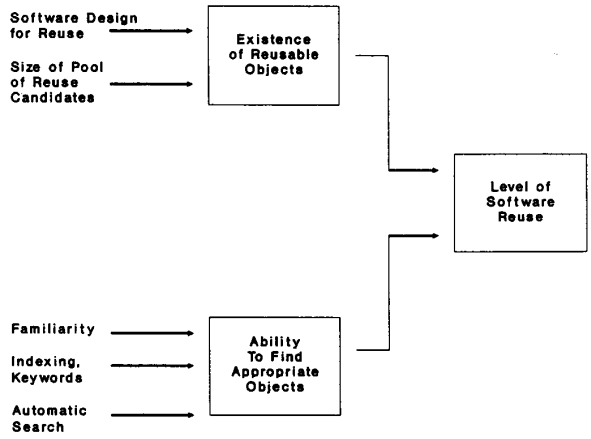


Fig. 3. A preliminary model of reuse.

rather than write a new one, depends upon the availability of potentially reusable software, and upon the programmer's ability to find it. We can increase reuse levels by making more reuse candidates available and by reducing programmers' search costs. This is the *software library* theory of reuse—that the keys to reuse are a large stock of objects within the application domain, and a catalog to help locate them as needed. HPS 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 incorporation of reused objects into the application system. From the perspective of the model in Fig. 3, this represents a strong foundation for a program of reuse.

Managers were aware of many of the limitations of these mechanisms, and of the relevance of organizational factors. They believed, correctly, that HPS's technical support of reuse could still allow them to realize far higher reuse levels than they had with traditional software development tools. The discussion that follows will seek to assess the utility of this approach, and to identify factors which will enable the achievement of yet higher levels of reuse.

The view of the reuse process depicted above suggests a number of predictions:

1) As the pool of reusable objects increases over time with the size of the repository, so will the level of reuse.

2) Objects belonging to the system currently being programmed are more likely to be known to the programmers, so they will exhibit comparatively low search costs, and there will be a relatively high level of internal reuse.

2a) 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 salient reusable objects and familiar reuse opportunities.

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

TABLE I
AN OVERVIEW OF THE HPS REPOSITORIES

Object Type	FBC	CHH
Rule Sets	8892	1775
Screens	7230	662
Domains	4200	97
Files	4236	170
3GL Modules	6062	92
Fields	6266	5823
Views	6755	3861

B. Repository Evaluation Findings

Automated repository analysis was used to assess each site's repository after about two years of HPS software developed. The two sites had very different startup experiences. CHH began using HPS two years later than FBC, when the tool was more mature. The analyses that follow skip the initial learning periods, and cover the 20 months following the first development successes. Table I gives an overview of the contents of the two repositories at the end of this time. The repositories reflect differences in the application domains. The retailer's systems, for example, may be seen to be far more data intensive.

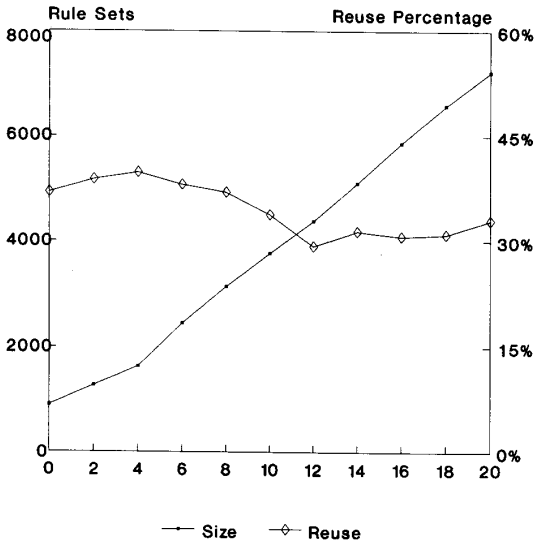
Repository Growth and Software Reuse: Fig. 4 presents the growth in Rule Set population and reuse during the periods under analysis.[5] It is immediately clear that our first prediction, that reuse levels would grow over time as the repository grew, was incorrect: the repositories grew steadily during this period. (So did the experience of the programmers, since this was their first experience with HPS.) Reuse percentage, however, achieved a strong initial value and never bettered it. The level of reuse didn't grow as the pool of reuse candidates grew.

Reuse of Familiar Objects: Our second prediction, which was based on the belief that familiar objects were more likely to be reused, was borne out strongly. We predicted that programmers would tend 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 at 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.

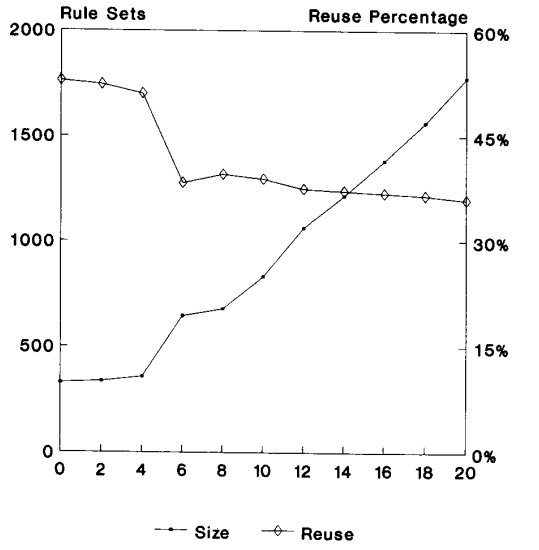
Fig. 5(a) shows the relationship between internal and external reuse: 85% of all observed instances of reuse were internal. That is, if use was made of a previously written rule, that rule was almost always one that had been written for the same system.

This offers a partial explanation of the leveling off of reuse over time. Reuse appears to be driven by the pool of a familiar code, rather than by the entire pool of reuse candidates. Each project is largely a self-contained universe (we assume that programmers will be most familiar with the code with which they are currently working than with

[5] Recall that Rule Sets are the most labor-intensive objects in these systems. 3GL modules might be more significant, except that they are typically used in cases where special-purpose routines are already "on the shelf."

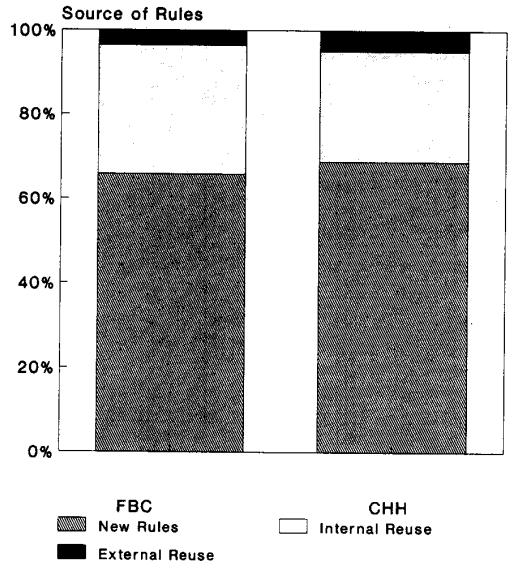


(a)

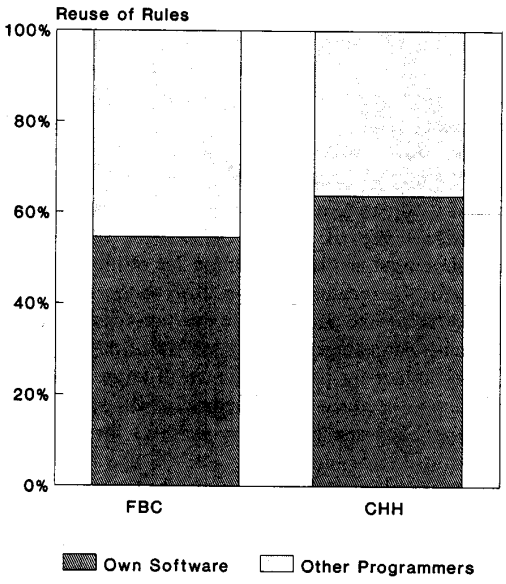


(b)

Fig. 4. Reuse and repository growth. (a) FBC. (b) CHH.



(a)



(b)

Fig. 5. (a) Internal and external reuse. (b) Reuse of own software.

that upon which other programming teams are working) and new projects derive little benefit from previous projects. The importance of familiarity suggests that the search mechanisms available are either inadequate or underutilized. As we explain below, we believe both to be the case.

Fig. 5(b) shows the prevalence of self-reuse. Despite the presence of over 250 programmers at FBC and over 100 programmers at CHH, more than 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. Fig. 6 shows the relationship between system size and reuse. The correlation between these two factors was 37% ($p = 0.09$) for 22 application systems at FBC and 58% ($p = 0.04$) for 13 application systems at CHH.[6]

The strong tendency of programmers to reuse objects of their own development is further evidence of the importance of familiarity. We are not able to estimate the degree to which the prevalence of internal reuse is also driven by two other

[6]Fig. 6 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.

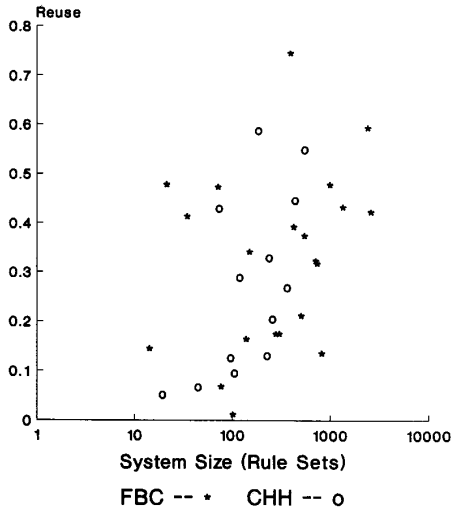


Fig. 6. Reuse and system size.

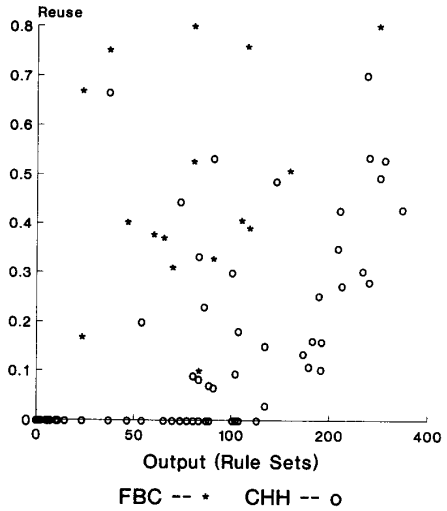


Fig. 7. Reuse and programmer output.

factors—the better "fit" an application system's own objects might be expected to have, and the efforts of the developers to design for internal reuse. We note, however, that the provision of a pool of generally reusable objects did not enable CHH to achieve higher levels of external reuse than FBC. This suggests that familiarity is at least as important a factor as fit.

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. Fig. 7 shows the distribution of programmer output and reuse. The top 5% of the programmers accounted for the creation of over 20% of the Rule Sets and for over 50% of the reuse, with the top reusers achieving average reuse percentages as high as 75%. 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 FBC and 60% ($p = 0.0001$) for ($n = 76$) at CHH.[7]

There are three possible explanations for these observations. One is that the same skills that make some programmers extraordinarily productive also make them extraordinarily good reusers of software. A second is that these programmers have a larger pool of familiar objects (i.e., objects of their own making) to reuse. A third is that we are observing an attitude change over time, with the high-reuse programmers simply being the ones who had been using HPS the longest, and had absorbed the reuse "message." The data did not bear this last hypothesis out: the partial correlations, controlling for months of HPS experience, were within 1% of the raw correlations.

In summary, it appears that HPS provides capabilities which allow programmers to achieve high levels of reuse. However, the pattern of reuse—with most reuse attributable to a small number of enthusiastic software reusers—suggests that there remains considerable unexploited reuse potential. Programmers are writing new objects rather than searching for reuse opportunities. It is of considerable interest to determine whether the high reuse levels achieved by the most productive programmers represent a skill that can be taught.

[7] Of the 110 programmers at CHH, only the 76 who wrote at least one Rule Set were included in this analysis. Our data for FBC represents a sample of 19 programmers out of 250. A log scale is used, for display purposes only, because order-of-magnitude differences in programmer outputs make a linear display difficult to interpret.

VI. Organizational Factors Affecting Software Reuse

In addition to analyzing the repositories, 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 software reuse opportunities. Most serious, however, were the organizational barriers and disincentives to reusing software.

A. Search and Organizational Incentives

HPS 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, however, 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.

The primary unexploited opportunity that we identified at FBC and CHH revolves around the lack of formal incentives to reuse objects. Managers believed that it was premature to enforce reuse benchmarks while they were still learning the best ways to use and to manage HPS and software reuse.[8] While formal incentives to reuse software were not

[8] In follow-up interviews at the sites, we learned that managers now believe that higher levels of reuse can result from a maturing managerial process based on formal productivity and reuse measurement. A study conducted at CHH by an independent outside consultant, subsequent to our study, disclosed that

a primary focus of management, informal incentives existed for a programmer to *prevent* others from reusing her objects. 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—often 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. Stronger change control mechanisms might have mitigated this problem, at the cost of interfering with the learning and experimentation that management was trying to encourage in its HPS programmers.

In practice, programmers who wish to use an object from another application are 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 a 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 adaptation costs are higher, and subsequent life cycle savings, particularly in maintenance, are not realized.

B. Preliminary Conclusions about Reusable Software

The initial drive for reuse at FBC and CHH 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 did not go far enough. In particular, they did not sufficiently stresses the organizational prerequisites for successful reuse. The repository-level analysis illustrated above heightened management awareness of organizational issues, and motivated a more complex model of software reuse.

The managers continue to believe that there are high degrees of commonality among the application systems at each site, but the relatively low levels of external reuse reinforce the importance of domain analysis, and formally designing for reuse, in achieving the full benefits of vertical reuse.

Fig. 8 presents a revised model of software reuse, in light of the repository evaluation results presented in Section V. The mostly technical factors that the earlier model presented as drivers of software reuse are still in place: 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 HPS. At the time this study was conducted, however, reuse appeared to have reached a plateau.

The immediate barriers to higher reuse levels appear to have been organizational. Software reuse was encouraged, but not mandated. Programmers were not rewarded for reuse while HPS use was still in the learning and innovation stage.

CHH now produces about 30% more function points per person-year at 30% less cost per function point, compared to a reference sample of over 25 other Fortune 500 companies. Management attributed this in part to its program of software reuse.

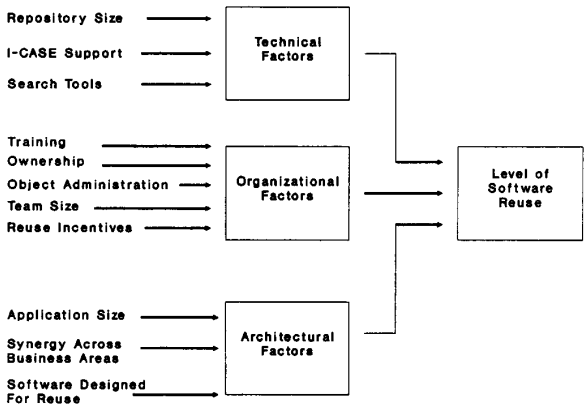


Fig. 8. A revised model of reuse.

The weakest technical aspect of HPS with respect to software reuse is the keyword search mechanism, which appears to be unequal to its task. Since most objects are relatively small, and since HPS is successful in making individual objects very easy to develop, programmers are willing to bear extremely low search costs before choosing to just write their own objects—in the absence of managerial incentives to search longer.

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 the realization of their full benefits requires corresponding changes in the software development process.

VII. Conclusion

Integrated CASE tools support not only the implementation of advanced software development processes, but also their monitoring and control. In this paper, we have used repository evaluation to study software reuse at two sites that are pursuing reuse by means of the same CASE tool. Repository evaluation allowed us to assess the success of these efforts. It enabled us to critique a simple model of software reuse, and to suggest a richer one.

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 nontechnical 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 programmers with the highest outputs of objects. We are investigating the question of whether this is a familiarity effect or a skill effect, as this would determine the best way to teach reuse.

We investigated the relative success of internal reuse, compared to that of external reuse. Our findings reinforced the messages of prior researchers, that success in external reuse cannot be achieved informally. It relies upon formal domain analysis and effective cataloguing and search mechanisms.

Repository evaluation allowed us to put numbers to aspects of reuse of which we previously had only a qualitative understanding—and it allowed senior management to assess the strengths and weaknesses of their software reuse efforts, and to decide how to improve them.

ACKNOWLEDGMENT

We wish to acknowledge M. Baric, G. Bedell, T. Lewis, and V. Wadhwa for the access they provided us to the software development activities and staff at The First Boston Corporation and Seer Technologies in New York City. B. Menar, N. Liebson, J. Yent, and D. Christy at Carter Hawley Hale Information Services, in Anaheim, CA, provided similar support. We also appreciated the research assistance of L. Erlihk, R. Kumar, and M. Oara, whose efforts to automate repository queries made this research possible. Finally, R. J. Kauffman thanks the Nippon Electric Corporation and the U.S.–Japan Business and Economics Research Center, Stern School of Business, New York University, for partial funding of data collection.

REFERENCES

[1] K. Allen, W. Krutz, and D. Olivier, "Software reuse: Mining, refining, and designing," in *TRI-Ada '90 Proc.*, Dec. 1990, pp. 222–226.

[2] U. Apte, C. S. Sankar, M. Thakur, and J. Turner, "Reusability strategy for development of information systems: Implementation experience of a bank," *MIS Quart.*, vol. 14, no. 4, pp. 421–431, Dec. 1990.

[3] R. D. Banker and R. J. Kauffman, "Reuse and productivity: An empirical study of integrated computer-aided software engineering (ICASE) at the First Boston Corporation," *MIS Quart.*, vol. 15, no. 3, pp. 375–401, Sept. 1991.

[4] ——, "Automated software metrics, repository evaluation and the software asset management perspective," Center Inform. Syst., Stern School of Business, New York Univ., Working Paper, 1991.

[5] ——, "Measuring the development performance of integrated computer-aided software engineering: A synthesis of field study results from the First Boston Corporation," in *Software Engineering Economics*, T. Gulledge, Ed. New York: Springer-Verlag, 1993, to be published.

[6] R. D. Banker, R. J. Kauffman, and Kumar, "An empirical study of object-based output metrics in a computer-aided software engineering environment," *J. Management Inform. Systems.*, vol. 6, no. 3, Winter 1992.

[7] R. D. Banker, R. J. Kauffman, C. Wright, and D. Zweig, "Automating output size and reuse metrics in a repository-based computer-aided software engineering environment," Stern School of Business, New York Univ., Working Paper, 1991.

[8] H. B. Barnes and T. Bollinger, "Making software reuse cost effective," *IEEE Software*, vol. 8, no. 1, Jan. 1991.

[9] V. Basili, "Viewing maintenance as reuse-oriented software development," *IEEE Software*, vol. 7, no. 1, pp. 19–25, Jan. 1990.

[10] B. Beckman, W. Van Snyder, S. Shen, J. Jupin, L. Van Warren, B. Boyd, and R. Tausworthe, "The ESC: A hypermedia encyclopedia of reusable software components," Jet Propulsion Lab., California Inst. Technol., Pasadena, CA, Sept. 1991.

[11] T. J. Biggerstaff and A. J. Perlis, Eds., *Software Reusability: Volume I—Concepts and Models*. New York: Addison-Wesley/ACM Press, 1989.

[12] ——, Eds., *Software Reusability: Volume II—Applications Experience*. New York: Addison-Wesley/ACM Press, 1989.

[13] B. A. Burton, R. W. Aragon, S. A. Bailey, K. D. Koehler, and L. A. Mayes, "The reusable software library," *IEEE Software*, vol. 4, no. 4, pp. 25–33, July 1987.

[14] G. Caldiera and V. R. Basili, "Identifying and qualifying reusable software components," *IEEE Computer*, pp. 61–70, Feb. 1991.

[15] M. J. Cavaliere, "Reusable code at the Hartford insurance group," in *ITT Proc. Workshop Reusability in Programming*, Newport, RI, 1983; reprinted in *Software Reusability: Volume II—Applications Experience*, T. J. Biggerstaff and A. J. Perlis, Eds. New York: Addison-Wesley/ACM Press, 1989.

[16] M. Chen and E. H. Sibley, "Using a CASE-based repository for systems integration," in *Proc. 1991 Hawaii Int. Conf. Systems Sciences*, IEEE, Jan. 1991, pp. 578–587.

[17] S. Cohen, "Process and products for software reuse in Ada," in *TRI-Ada '90 Proc.*, Dec. 1990, pp. 227–239.

[18] M. Cusamano, *Japan's Software Factories: A Challenge to U.S. Management*. Oxford, England: Oxford University Press, 1991.

[19] E. M. Dusink, "Towards a design philosophy for reuse," in *Proc. Reuse in Practice Workshop*, J. Baldo and C. Braun, Eds. Pittsburgh, PA: Software Eng. Inst., 1989.

[20] J. B. Frakes, T. J. Biggerstaff, R. Prieto-Diaz, K. Matsumura, and W. Shaefer, "Software reuse: Is it delivering?" in *Proc. 13th Int. Conf. Software Eng.*, Austin, TX, IEEE Comput. Soc. Press, May 13–17, 1991, pp. 52–59.

[21] J. B. Frakes and Nejmeh, "Software reuse through information retrieval," in *Proc. 20th Hawaii Int. Conf. Syst. Sci.*, B. D. Shriver and R. H. Sprague, Jr., Eds., Kailua-Kona, HI, 1987, pp. 530–535.

[22] P. Freeman, Ed. *Tutorial on Software Reusability*. Washington, DC: IEEE Comput. Soc. Press, 1987.

[23] J. E. Gaffney, Jr. and T. A. Durek "Software reuse—Key to enhanced productivity: some quantitative models," *Inform. Software Technol.*, vol. 31, no. 5, pp. 258–267, June 1989.

[24] J. W. Hooper and R. O. Chester, *Software Reuse: Guidelines and Methods*. New York: Plenum, 1991.

[25] W. S. Humphrey, *Managing the Software Process*. Reading, MA: Addison-Wesley, 1989.

[26] R. G. Lanergan and C. A. Grasso, "Software engineering with reusable designs and code," *IEEE Trans. Software Eng.*, vol. SE-10, no. 5, pp. 498–501, Sept. 1984.

[27] M. Lenz, H. A. Schmid, and P. F. Wolfe, "Software reuse through building blocks," *IEEE Software*, vol. 4, no. 4, pp. 34–42, July 1987.

[28] D. G. McNichol, C. Palmer, S. G. Cohen, W. H. Whitford, and G. O. Goeke, "Common Ada missile packages—CAMP, Vol. I: Overview and commonality study results," McDonnell Douglas, St. Louis, MO, Tech. Rep. AFATL-TR-85-93, 1986.

[29] B. McNurlin, "Building more flexible systems," *I/S Analyzer*, Oct. 1989.

[30] Y. Matusmoto, "Some experiences in promoting reusable software: Presentation in higher abstract levels," *IEEE Trans. Software Eng.*, vol. SE-10, no. 5, pp. 502–512, Sept. 1984.

[31] B. Meyer, *Object-Oriented Software Construction*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

[32] J. M. Neighbors, "The DRACO approach to constructing software from reusable components," *IEEE Trans. Software Eng.*, vol. SE-10, no. 5, pp. 564–574, Sept. 1984.

[33] W. E. Novak, "U.S. Army SDS CRWG reuse committee technical requirements document: Technical guidance section," draft version, 1990.

[34] J. F. Nunamaker, Jr. and M. Chen, "Software productivity: A framework of study and an approach to reusable components," in *Proc. 22nd Hawaii Int. Conf. Syst. Sci.*, IEEE, Jan. 1989, pp. 959–968.

[35] ——, "Software productivity: Gaining competitive edges in an information society," in *Proc. 22nd Hawaii Int. Conf. Syst. Sci.*, IEEE, Jan. 1989, pp. 957–958.

[36] A. Pollack, "The move to modular software," *New York Times*, pp. D1–D2, Apr. 23, 1990.

[37] Polster, "Reuse of software through generation of partial systems," *IEEE Trans. Software Eng.*, vol. SE-10, no. 5, pp. 402–416, Sept. 1984.

[38] R. Prieto-Diaz, "Classifying software for reusability," *IEEE Software*, vol. 4, no. 1, pp. 6–16, Jan. 1987.

[39] ——, "Domain analysis: An introduction," *ACM Software Eng. Notes*, vol. 15, no. 2, pp. 47–54, Apr. 1990.

[40] H. D. Rombach, "Software reuse: A key to the maintenance problem," *Inform. Software Technol.*, vol. 33, no. 1, Jan./Feb. 1991.

[41] J. A. Senn and J. L. Wynekoop, "computer-aided software engineering (CASE) in perspective," Inform. Technol. Management Center, College Business Administration, Georgia State Univ., Working Paper, 1990.

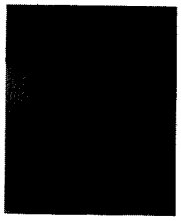
[42] *CASE Research Report*, Sentry Market Research, Westborough, MA, 1990.

[43] V. Sepannen, "Reusability in software engineering," in *Tutorial: Software Reusability*, P. Freeman, Ed. Austin, TX: IEEE Comput. Soc. Press, 1987, pp. 286–297.

[44] "Software productivity metrics working group of the software engineering standards subcommittee, standards for software productivity metrics," IEEE Comput. Soc. P1045/D5.0 (draft), Mar. 8, 1992.

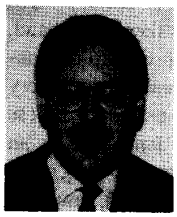
[45] W. Tracz, "RECIPE: A reusable software paradigm," in *Proc. 20th Hawaii Int. Conf. Syst. Sci.*, B. D. Shriver and R. H. Sprague, Jr., Eds., Kailua-Kona, HI, 1987, pp. 546–555.

[46] ——, *Tutorial: Software Reuse—Emerging Technology*. Austin, TX: IEEE Comput. Soc. Press, 1988.



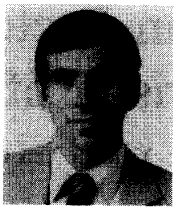
Rajiv D. Banker received the Doctorate in business administration from Harvard University, Cambridge, MA, in 1980, with a concentration in planning and control systems.

He is the Arthur Andersen & Co./Duane R. Kullberg Chair in Accounting and Information Systems at the Carlson School of Management, University of Minnesota, Minneapolis. He has published numerous articles and serves on the editorial boards of several prominent research journals. His research on information systems development and maintenance is field-based and empirical, involving collection and analysis of data on software complexity, project characteristics, systems environment and programmer experience, and ability and effort, to estimate the impact of managerial and technological factors on productivity and quality of commercial software. Of particular interest is the study of integrated CASE technologies and the management of reusable software.



Robert J. Kauffman received the Doctorate in industrial administration from Carnegie Mellon University, Pittsburgh, PA.

He is an Associate Professor of Information Systems at the Stern School of Business, New York University (on leave in 1992–1993 at the Federal Reserve Bank of Philadelphia and the Simon Graduate School of Management, University of Rochester), where he specializes in information technology in the financial services sector. Previously an international bank lending and strategic planning officer, he is currently Nippon Electric Corporation (NEC) Faculty Fellow of the U.S.–Japan Business and Economics Research Center. He has published articles in *MIS Quarterly, Journal of Management Information Systems*, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, *Information and Software Technologies, Information and Management*, and elsewhere. His current research focuses on developing new methods for measuring the business value of information technology investments (including CASE), using techniques from finance and economics.



Dani Zweig received the Doctorate in industrial administerial administration from Carnegie Mellon University, Pittsburgh, PA.

He is an Assistant Professor of Information Systems in the Department of Administrative Sciences at the Naval Postgraduate School, Monterey, CA. Prior to this he was a consultant for Peat Marwick. His research focuses on software reuse and on cost implications of software complexity. He is also currently working on an analysis of the Department of Defense's software inventory, its rate of obsolescence, and expected replacement costs.