TOOLS FOR MANAGING REPOSITORY OBJECTS

Rajiv D. Banker
Carlson School of Management
University of Minnesota

Tomas Isakowitz
and
Robert J. Kauffman
Stern School of Business
New York University

Rachna Kumar
College of Business Administration
University of Texas at Austin

Dani Zweig
Department of Administrative Sciences
Naval Postgraduate School

Working Paper Series
STERN IS-93-46

TOOLS FOR MANAGING REPOSITORY OBJECTS

Rajiv D. Banker
Carlson School of Management
University of Minnesota

Tomas Isakowitz
and
Robert J. Kauffman
Stern School of Business
New York University

Rachna Kumar
College of Business Administration
University of Texas at Austin

Dani Zweig
Department of Administrative Sciences
Naval Postgraduate School

1. AUTOMATING REPOSITORY EVALUATION

The past few years have seen the introduction of repository-based computer aided software engineering (**CASE**) tools which may finally enable us to develop software which is reliable and affordable. With the new tools come new challenges for management: Repository-based CASE changes software development to such an extent that traditional approaches to estimation, performance, and productivity assessment may no longer suffice — if they ever did. Fortunately, the same tools enable us to carry out better, more cost-effective and more timely measurement and control than was previously possible.

Automated Metrics and the Management of an Object Repository

From the perspective of senior managers of software development, there are three characteristics of the new technologies that stand out:

(1) *Productivity enhancement.* Development tasks that used to require great effort and expense may be largely automated. This changes the basis for software cost estimation and control.

(2) *Software reuse.* The repository acts as a long-term storehouse for the firm's entire application systems inventory. It stores it in a manner which makes reuse more practical. Firms that hope to achieve high levels of reuse (on the order of 50%) must move from generally encouraging reuse to explicitly managing it.

(3) *Access to measurement.* The repository holds the intermediate lifecycle outputs — of analysis and design, and not just the final software product. As a result, it becomes practical to automate the computation of the metrics which managers need in order to take full advantage of the new technologies.

Over the last several years, we have been conducting a research program to shed light on how integrated CASE supports improved software productivity and software reliability through the reuse of repository software objects. We have found that successful management of this effort depends upon a number of factors:

(1) the reliability of cost estimation for CASE projects, in an environment in which source lines of code are almost meaningless, and in which costs can vary by a factor of two depending on the degree of reuse achieved;

(2) the extent to which software developers effectively search a repository to identify software objects that are candidates for reuse;

(3) how software reuse is promoted and monitored; and,

(4) the extent to which various kinds of software objects (especially those which

are the most expensive to build) are actually reused.

Managers can only hope to control these factors if they can measure them, and measure them in a cost-effective manner. In practice, this means automating as much of the analysis as possible. Fortunately, our research has shown that it is feasible to do so -- and to a far greater extent than we initially envisioned. By automating a number of useful repository evaluation procedures, we can provide senior managers with new perspectives on the performance of their software development operations.

STRESS: Seer Technologies Repository Evaluation Software Suite

Our long-term study of CASE-based software development continues at several sites that deployed the same integrated CASE tools. Among them are The First Boston Corporation, a New York City-based investment bank, and Carter Hawley Hale Information Services, the data processing arm of a large Los Angeles-based retailing firm. These firms allowed us to examine extensively and report on their evolving software object repositories. (For a more detailed discussion of these studies, see [1] and [2].) Their repositories were created with an integrated CASE tool called *High Productivity Systems* (**HPS**). HPS promotes modular design, object reuse and object naming conventions. It also enables the programming of applications that can be run cooperatively on multiple operating platforms, without requiring a developer to write code in the programming language that is native to each of the platforms. Instead, HPS simplifies development, by enabling the developer to create software functions using a single fourth generation "rules language", which is then processed by a code generator and translated into whatever 3GL source languages best suit the target platforms.

The metrics which we, as researchers, needed in order to analyze software development were the same ones that the managers needed in order to control it. The primary insight which made the measurement practical was that all the information that was needed could be derived from information that was already stored within the repository. In cooperation with The First Boston Corporation and Seer Technologies (the original developers of HPS), we began to develope the conceptual basis of *STRESS, Seer Technologies' Repository Evaluation Software Suite*, a set of automated software repository evaluation tools. At present, STRESS consists of several automated analysis tools:

(1) **FPA**, the automated *Function Point Analyzer*,

(2) **OPAL**, the *Object Points Analyzer*, a new software cost estimation capability

(3) **SRA**, the automated *Software Reuse Analyzer*,

(4) **ORCA**, the *Object Reuse Classification Analyzer*.

The remainder of this paper describes the STRESS tool set in greater detail, and discusses how it can make repository object management possible.

2. FUNCTION POINT ANALYSIS

The most commonly-used bases for estimating and controlling software costs, schedules, and productivity are *source lines of code* and *function points*. The function point methodology, which computes a point score based on the functionality provided by the system, is illustrated in Figure 1. A standard weight is assigned to each system function, based on its type and complexity (e.g., 5 points for an output of average complexity), and the total count is multiplied by an environmental complexity modifier which reflects the impact of task-specific factors.

Function point analysis, which measures the amount of data processing actually being performed by a system, has a number of advantages over counting source lines of code. Function points are language-independent, they allow for differences in task complexity between systems of similar size, and they can be estimated much earlier in the life cycle. For example, we can estimate function points during design, when we know what the system will do, but source lines of code can't physically be counted until the end of the coding phase.

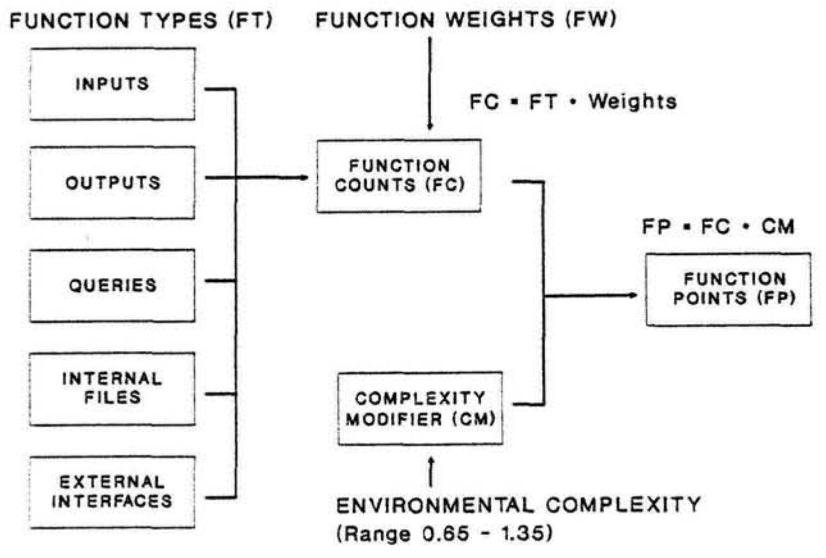


Figure 1: Function Point Analysis

Despite these benefits and others, source lines of code remain the more commonly used measure. Function point analysis requires considerable and expensive manual effort to compute, whereas the counting of source lines is easily automated. For integrated CASE environments such as HPS, however, counting source lines of code is of relatively little use: much of the functionality of the system is represented in the CASE tool's internal representation, rather than in traditional source code. Our solution was to use that internal data in automating the function point analysis.

The Function Point Analyzer (FPA)

Function point analysis has been difficult to automate in traditional software engineering environments, because it requires detailed knowledge of the system being analyzed. For example, the analyst must know whether the module which will receive a data flow is considered to be part of the application system or external to it. This information may not be readily available, but it will determine whether the data flow counts towards the system's function point total. Or, the analyst must know how many data elements are being passed within a given data flow, as this will determine the complexity, and hence the value, of that function point contribution. Or, the analyst may have to examine the code of a data input module to make sure that the designers didn't use the same module to also perform some output function (e.g., display prompts), which might count towards the function point total. Even if such information is available in the system documentation (as in principle it ought to be, and in practice it often is not), the number of such decisions which have to be made add up to a formidable amount of paper-chasing for the analyst.

In an integrated CASE environment, most or all of this information will already be contained within the repository. The information which an integrated CASE tool must store about the system whose development it is supporting includes much or all of the information needed for the function point analysis. Different CASE tools will store the information in different ways. Figure 2 illustrates the mapping from the HPS repository representation of a software application to its equivalents in user functions.

The objects inside the application boundary on the figure are those which belong to the system being analyzed, and the lines connecting them represent calling relationships. In traditional systems, the analyst must rely upon naming conventions to determine which modules belong to a system and which don't. The analyst may also have to examine the actual code so as not to be misled by, for example, software reuse or obsolete documentation. In the integrated CASE environment, each calling relationship between a pair of objects is stored in the repository as part of the tool's knowledge about the system. The *Function Point Analyzer* (FPA) can identify the objects which are part of the application system by searching the repository.

Similarly, the repository has to know precisely what data elements are being passed to or from each and every object, in order to maintain the control and consistency needed by an integrated environment. The Function Point Analyzer can

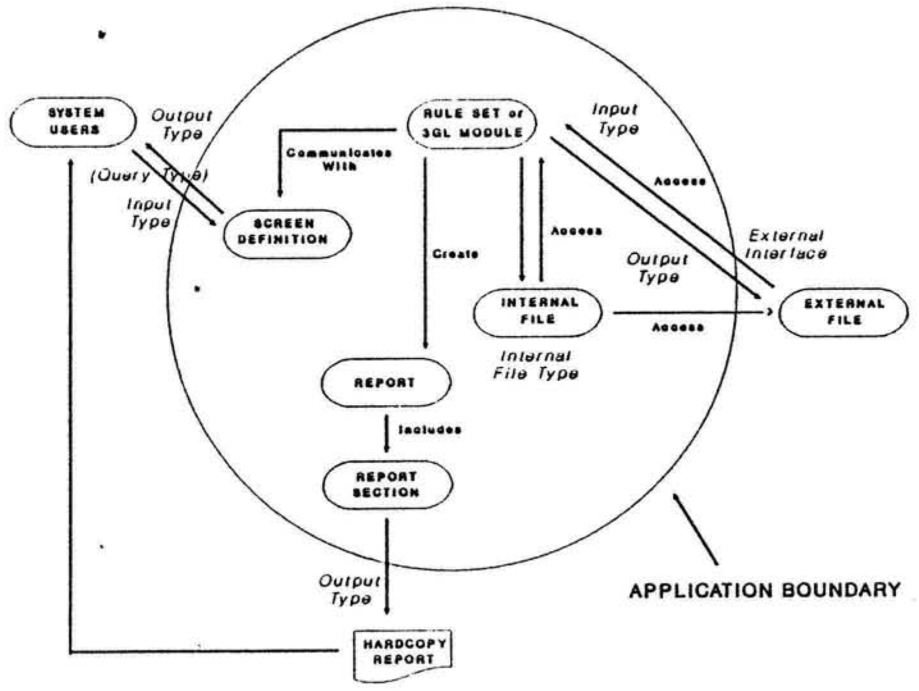


Figure 2: Mapping HPS Objects to Function Points

Determining the actual functionality of each object is the most implementation-dependent step, and the one that will vary the most from CASE tool to CASE tool. In HPS, the semantics of the 4GL *Rules Language* (a meta-language representing the objects and calling relationships that define the functionality of an application) constrain each object to a well-defined purpose (e.g., controlling one window, or generating one report segment). Since all interactions between HPS objects are mediated by database 'views', and since all database views are in the repository, the Function Point Analyzer can read the type and complexity of each data flow directly from the repository.

What all these capabilities of the Function Point Analyzer have in common is that they only depend on the information which HPS maintains, internally, *about* the system. At no point does it become necessary to examine the code itself.

3. OBJECT POINT ANALYSIS

Automating function point analysis gave us a good basis for tracking productivity improvements, both against the firms' old baselines, and against industry standards and industry leaders. Interviews with project managers, however, revealed that there were disadvantages to using Function Points as a basis for controlling individual HPS projects:

(1) Function points collapse the benefits of enhanced productivity through CASE-based automation and the benefits of software reuse.

(2) The shift to CASE was accompanied by a growing emphasis on early-life-cycle activities, particularly enterprise modelling and business analysis, and function points are more oriented towards design and post-design activities.

(3) HPS developers and managers were used to working directly with HPS objects, and the mapping from objects to function points wasn't intuitive to the managers. For the first time, the mapping was close enough that managers could think of asking for better. What they wanted was a way to use the repository objects directly, as a basis for planning and control.

In order to satisfy this demand, we had to first develop an estimation mechanism that was based on repository objects, and then demonstrate that it could equal function point analysis in predictive power and automatability.

The Object Point Analyzer (OPAL)

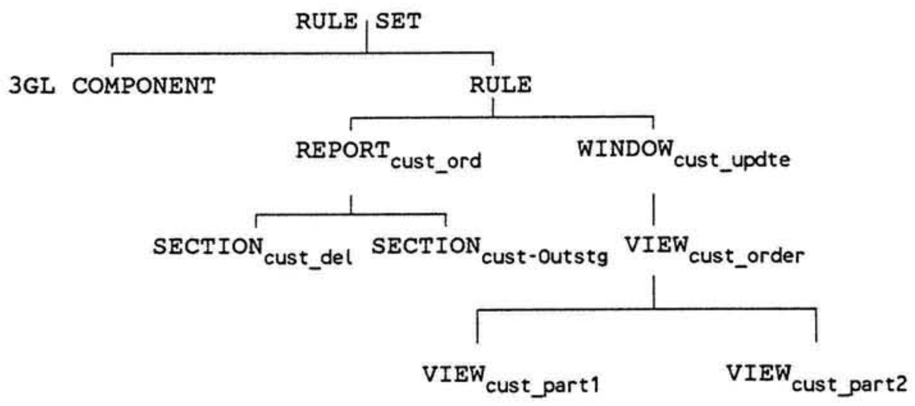
In an integrated CASE environment, the repository objects created in early phases of the software development life cycle will be high-level abstractions of those to be created during the coding/construction phase. The more information the repository contains about those early objects, the better our ability to make early and reliable predictions of project costs. As was the case with the function point analyzer, the specifics of the mapping will depend upon the implementation of the CASE environment.

OPAL, the *Object Point AnaLyzer*, was developed as a cost estimation facility for the HPS environment. It differs from the Function Point Analyzer primarily in providing a direct mapping from HPS objects to cost estimates.

Our interviews with project managers revealed that they were already using object-based cost estimation informally, assigning so many days of development effort for each type of object. Using those informal heuristics as a starting point, we used regression analysis first to give us more precise estimation weights and later to validate these results against actual projects.

OPAL computes *object points*, a metric inspired by function points, but better suited to the ICASE environment. Object points are based directly upon the objects stored within the repository, rather than upon the interactions between those object. In HPS terms, object points are assigned for each WINDOW, for each REPORT, for each 3GL MODULE, etc. Instances of each object type can be simple, average, or complex, with the more complex objects receiving higher object point scores. The computation of object points is illustrated in Figure 3. The objects depicted are part of a much larger application. Each object is assigned a complexity rating, based on empirically derived factors such as the number of objects it calls in turn, and then an object point score.

Because the CASE environment limits the functionality allowed to each object type, this is a true measure of application system functionality as well as of programmer effort. It was practical to automate the classification of objects because of the information the repository maintains about each object. Like FPA, OPAL uses the repository's internal representation of the application system to determine which objects should be considered in the analysis, and what complexity ratings to assign each object. The corresponding effort estimates are taken from OPAL's object-effort-weight tables. These store standard cost estimates, derived through prior empirical analysis, for simple, average, and complex instances of each object type.



OBJECT TYPE	COMPLEXITY DEFINING CHARACTERISTICS	COMPLEXITY CLASS	OBJECT POINTS
WINDOW	3 VIEWS; 1 RULE	average	2
REPORT	2 SECTIONS; 1 RULE	simple	2
3GL COMPNT		complex	8
...			

Figure 3: Illustration of Object Point Computation

We used nineteen medium-to-large software development projects to test OPAL's cost estimates against those based on function point analysis. The two estimators were found to be equally good predictors, but managers found object points easier to use and to interpret.

The results of the object point analysis can be presented in various ways, according to the requirements of the manager. Figure 4, for example, gives an object point breakdown of a subsystem, by object type.

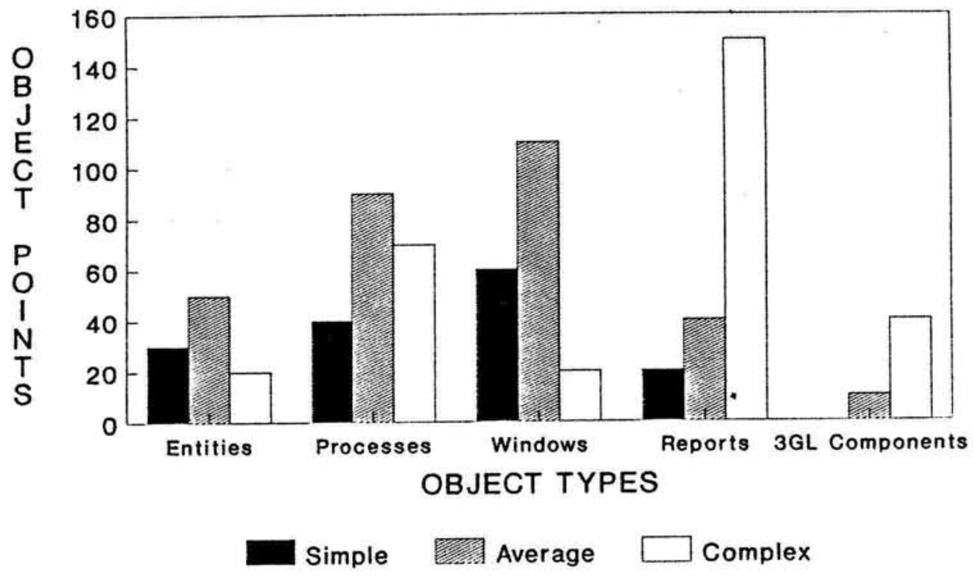


Figure 4: Object Point Breakdown by Object Type for Subsystem "Reorder"

4. SOFTWARE REUSE ANALYSIS

Software reuse is known to be a major source of productivity gains in software development. Based on claims that are often seen in the popular press, some organizations routinely expect reuse levels of 30 to 50%. But such high levels of reuse require an environment in which software reuse is supported from both a technical and a managerial standpoint; appropriate incentives for developers to reuse software; and a measurement program that provides a feedback mechanism to tell developers how much their efforts are paying off.

Object-based integrated CASE tools such as HPS provide the requisite technical support: they store software objects at a level of granularity which is far more conducive to reuse than traditional procedure-based software. They may also automate the mechanics of implementing reuse. HPS, for example, allows developers to reuse an object by simply adding a calling relationship to the repository. Measurement of reuse is also possible with CASE, especially when there is a repository that stores objects and their calling relationships. Such measurement is a prerequisite for accurate cost estimation. After all, software project cost estimation isn't going to be very reliable if we don't know whether to expect 30% reuse or 70% reuse of pre-existing software in a new system!

The Software Reuse Analyzer (SRA)

The repository-based architecture of HPS makes it practical, as we saw in the description of the function point analyzer, to query the repository to determine the extent of software reuse. This is accomplished through SRA, the *Software Reuse Analyzer*, which begins its analysis by creating a list of objects belonging to a given system. (This part of the analyzer's software was first developed for FPA and then, appropriately enough, reused in SRA.) We can also query the repository to determine how many times each object has been reused. Finally, the CASE tool maintains an object history which allows us to distinguish between internal reuse and external reuse. *Internal reuse* occurs when an object is created for a given application system and then used multiple times within that system. *External reuse*, on the other hand, occurs when the object being reused was initially created for a different application. The latter is more difficult to achieve, but is also more profitable.

SRA was built to deliver a number of useful managerial metrics. For example, it reports on two related metrics that offer an at-a-glance picture of the extent of reuse in an application: *new object percent*, the percentage of an application that had to be custom-programmed, and *reuse percent*, the percentage of the application constructed from reused objects. As we pointed out above, managers will further wish to distinguish between internal and external reuse percentages, to gauge how effectively developers are leveraging the existing repository. SRA can decompose reuse percent into *internal reuse percent* and *external reuse percent*.

A second important piece of information that managers will want is the business value of software reuse that is occurring. This is captured by SRA in a metric called *reuse value*. Reuse value is computed by translating the standard cost of the effort that would have been required, had the software objects that were reused been built from scratch. This is a highly useful metric because it helps managers to determine whether reuse pays off in development cost reductions.

Management of Software Reuse

SRA may be used to track software reuse within a given project, but such analysis generally comes after the fact. The main power of the tool is guiding the organization's long-term software reuse efforts. Figure 5, shown below, tracks our two sites' software reuse efforts over a comparable 20-month period.

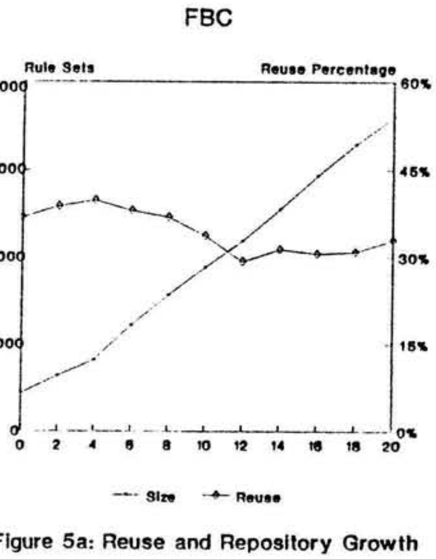


Figure 5a: Reuse and Repository Growth

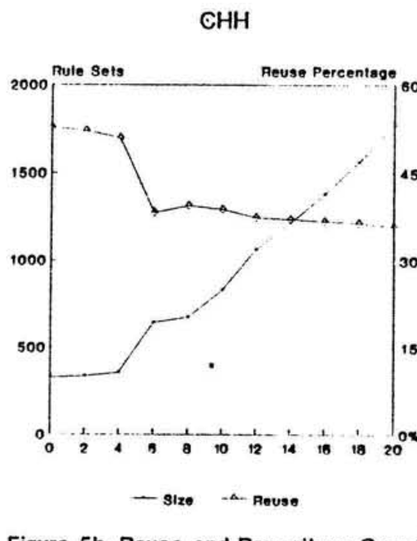


Figure 5b: Reuse and Repository Growth

The striking result is that while repository sizes grew steadily throughout the observation period, reuse levels almost immediately stabilized around the 30% level. Further use of SRA enabled us to analyze these results. Since HPS maintains a repository history of each object, it was possible to determine who created and who reused each object, and for which applications. The results were enlightening, as suggested by Figures 6 and 7 below.

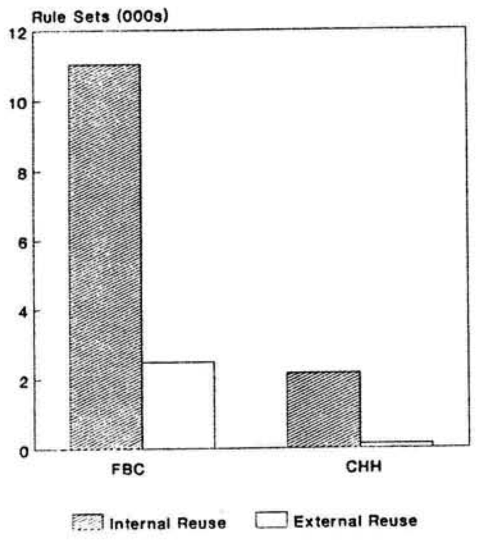


Figure 6: Internal and External Reuse

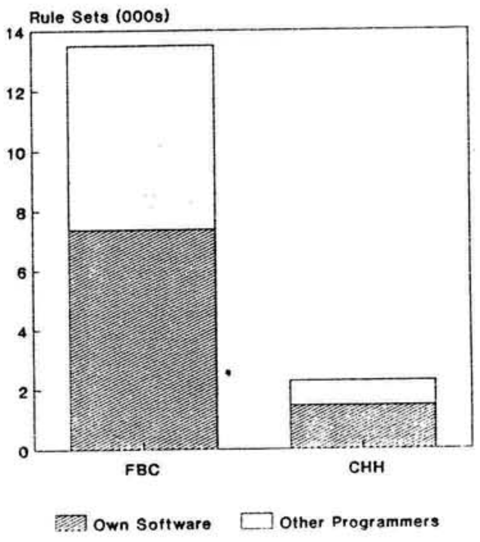


Figure 7: Reuse of Own Software

There was a strong and expected bias towards internal reuse. Developers preferred to get as much leverage as possible from the objects of the system under construction, rather than search the other systems in the repository for reuse candidates. What was not expected, however, was that most of the instances of external reuse consisted of programmers reusing objects that they themselves had previously created. In other words, little effort was being made to search for reusable objects. If developers personally knew of a reuse candidate, they used it; if not, it was simpler to write a new object than to search the repository for a reusable one. This went a long way towards explaining why the growth in repository size, and hence in reuse opportunities, was not resulting in growth in the reuse rates.

Figure 8 graphs the reuse levels of individual programmers against their overall output.

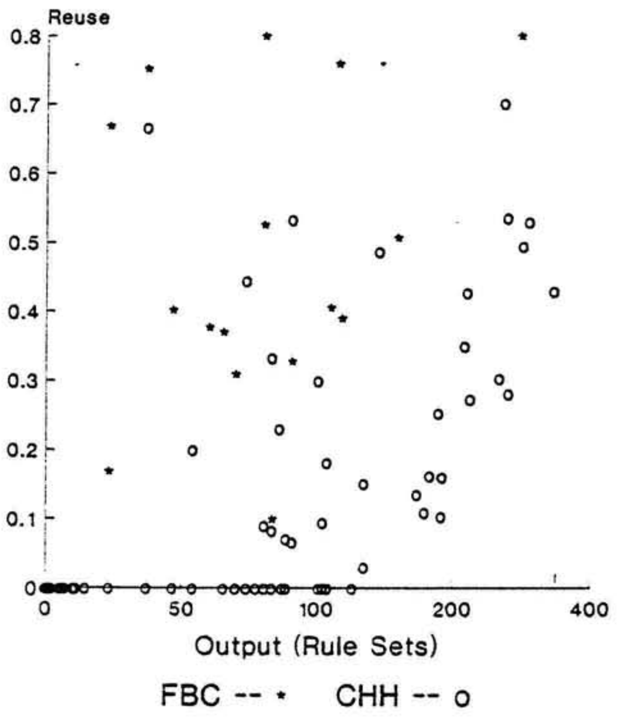


Figure 8: Reuse and Programmer Output

What the wide variation in programmer performance tends to obscure is the impact of the extremes. Software reuse analysis revealed that over 50% of the programmers at the research sites contributed no reuse whatsoever. On the other hand, the top 5% were responsible for over 20% of the objects in the repository and over 50% of the reuse. Only a few programmers were taking advantage of the substantial productivity gains that software reuse offered.

5. OBJECT REUSE CLASSIFICATION

We conclude our overview of the STRESS toolset with a discussion of a tool which is still in the research and development phase: the *Object Reuse Classification Analyzer.* Whereas software reuse analysis measures the level of reuse achieved, object reuse classification enables us to determine the repository's reuse potential, and supports developers in achieving that potential.

Repository Search for Reuse

We observed that one of the striking results of the software reuse analysis was the propensity of developers to reuse familiar objects, rather than to search extensively for unfamiliar, but possibly superior, reuse candidates. A mature repository may easily contain tens of thousands of software objects, only a fraction of which will be familiar to any one programmer or analyst. A developer who focuses on familiar objects (and most of the reuse we observed involved developers reusing software they themselves had created) will miss many software reuse opportunities.

Our interviews with HPS programmers confirmed what others have already discovered: search is difficult. The high productivity of an integrated CASE environment such as HPS makes it faster to write a new object from scratch than to search an enormous repository for an existing object which is a close enough fit. (This is as true for the analyst trying to design a system which will take advantage of software reuse as it is for the programmer trying to find an object to perform a specific task.) A more extended search may pay long-run dividends, in the form of reduced maintenance costs, but this is an argument which programmers and project managers have rarely found convincing in the face of immediate schedule pressures. So, if we want developers to take advantage of the untapped reuse potential, we have to provide automated search support.

Figure 9 illustrates the conceptual foundation of object classification analysis. We can think of the repository as consisting of a large number of objects within a "search space", with similar objects being closer together and dissimilar objects being further apart. The classification scheme is used to produce a similarity metric that determines the "distance" between repository objects. We can then give the system a description of the object we need, and ask for a short list of repository objects which are 'close' enough to the described object to be reuse candidates.

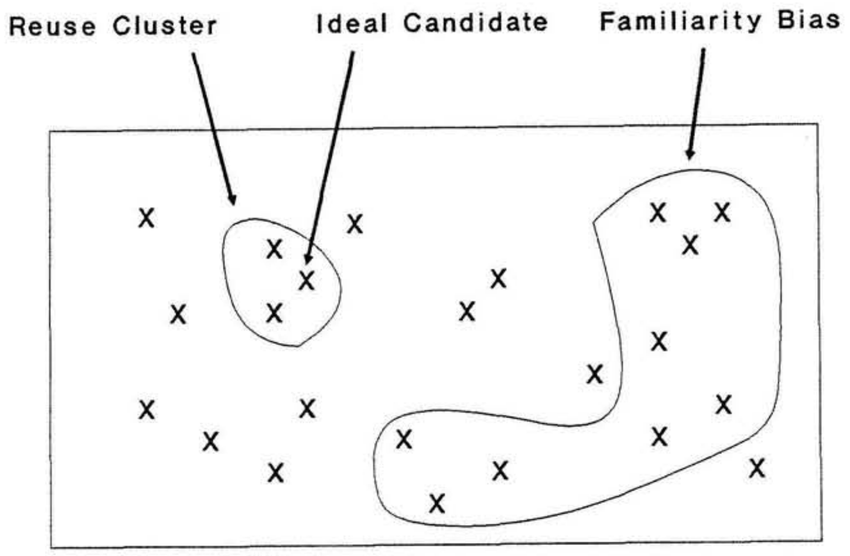


Figure 9: Reuse Clustering

Object Reuse Classification Analyzer (ORCA)

ORCA, the Object Reuse Classification Analyzer, has three functions: classification support, development support, and repository evaluation support.

1) **Classification support.** The classification scheme used by ORCA is an extension of Prieto-Diaz's faceted classification schema [4]. In such a schema, an object is classified along a number of dimensions — the facets -- and two objects may be 'close' to each other with respect to one or more facets. Figure 10, for example, illustrates a four-facet classification of a needed software module, and of two candidates for reuse. In this example, the functional similarities between the first component and the target object make it a better candidate than the second component, even though the second component was written for the target setting.

	NEEDED	COMPONENT 1	COMPONENT 2
Function	cross-validate	cross-validate	purge
Application	personnel	inventory	payroll
Objects	dates	dates	records
Setting	bank branch	dept store	bank branch

Figure 10: A Four-Facet Classification of Software Entities

ORCA supports *multiple classification criteria*. Multiple sets of facets may be defined, instead of a single criterion, or a single set of facets, with different classifications applying to different object types and to different stages of software development. Each set presents a criterion by which to analyze the repository. This allows, for example, for the case of two objects which would be judged to be far apart during business design, but might be closely related during technical design. The technical functionality may be similar, even when the business application functionality appears to be unrelated. Based on this multi-faceted classification schema, we can compute a quantitative metric to determine functional similarity between objects.

As the classification example suggested, an object classification scheme will use a combination of technical characteristics (e.g., object type, application system) and functional characteristics (e.g., purpose of module). The technical characteristics can be determined automatically, from information in the repository. For other facets, the developer can be prompted to choose from a list of options. The specific functionality-related classes and options may differ from one site to another, in which case the schema must be customized on the basis of interviews with software developers.

2) **Development Support.** The key design principle is to reduce the developer's involvement in the screening stage to a minimum — to let the analyzer worry about finding the potential needles in the haystack — and to

provide a short enough list of candidates that the developer will be able to give serious consideration to each. The search for reuse candidates takes place in two stages:

* Stage 1, *screening*, involves the purposeful evaluation of a large set of object reuse candidates from the entire repository to produce a short list of near matches for further investigation.

* Stage 2, *identification*, enables the developer to examine individual objects more closely to determine whether there is a match in terms of the required functionality.

When systems design is done well, it is very likely that a by-product of the effort will be a repository representation which can be matched to other existing repository objects at the time that technical design is completed. What remains is to ensure that there is a mechanism in place that enables a designer to test his design against the existing repository to determine what functionality might be reused as is, what might be adapted from very similar objects, and what needs to be built from scratch.

3) **Repository Evaluation Support**. Besides helping developers to find and inspect candidates for reuse, ORCA may also be used to classify objects and evaluate the repository as a whole. On the one hand, it can be used to identify redundancy -- unexploited reuse opportunities. A mapping of the repository will identify "reuse clusters", sets of objects which are similar in functionality, and can probably be consolidated into a smaller number of objects. Figure 11 illustrates the results of such consolidation.

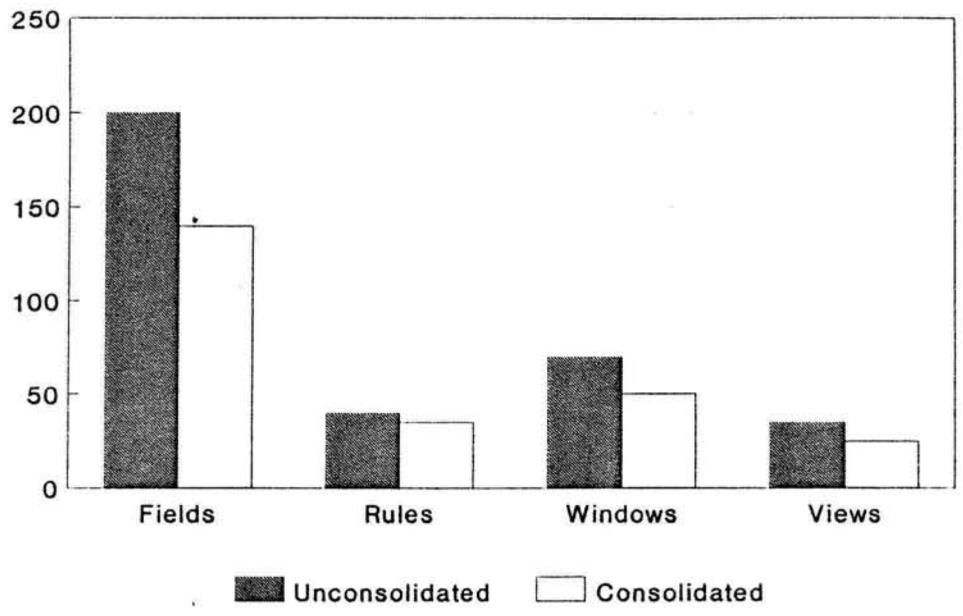


Figure 11: Consolidating a Repository by Detecting Reuse Clusters

Note that different types of repository objects are likely to benefit differently from such consolidation. On the other hand, such a mapping may identify gaps in the repository -- application areas in which developers will be less able to rely upon reuse support from the rest of the repository.

6. TOOLS TO MANAGE THE REPOSITORY: A RESEARCH AGENDA

Our current research efforts on repository object management software tools are focused on four primary tasks:

(1) *Implementation of the tools to support measurement will support longitudinal analysis of productivity and reuse.* With the help of Seer Technologies, we are working to install the Function Point Analyzer and Software Reuse Analyzer at a number of firms, in the U.S., Europe and Asia. This will enable us to carry out a large-scale longitudinal study of development productivity and software reuse, that expands upon our pilot studies in these

domains. As Kemerer [3] has pointed out, one of the most challenging problems facing software development managers is how to speed the move down the CASE development learning curve. In the absence of empirical results that estimate the learning curves that different firms have actually experienced, it will be difficult to provide much guidance as to the factors that enhance or inhibit firms to achieve better performance more rapidly.

(2) *We plan to further examine opportunities to extend the capabilities of the repository evaluation software tools to support other kinds of analysis.* We have already done a significant amount of this work on an informal basis, through specially developed repository queries. These queries have enabled us to investigate aspects of the repository that help to explain the 30% technical cap on reuse that we observed in the early days of software development at The First Boston Corporation and at Carter Hawley Hale Information Services. They also allowed us to determine which developers reuse software objects the most, and what kinds of software objects are involved. The results of such analysis has provided senior management at the firms whose data we analyzed with a fresh perspective on their software development operations.

(3) *The object points concept requires further empirical research to validate it for use in multiple settings.* Additional field study work, with Seer Technologies and its clients, and with other CASE vendors and their clients, will enable us to apply and validate the object point metrics we have proposed for software cost estimation in repository object-based integrated CASE environments. This process will only be possible through the deployment and application of the Object Point Analyzer, OPAL. We expect that additional field study research will enable us to uncover the extent to which the object complexity weights may vary with different software development environments.

(4) *Additional conceptual and empirical research is required to support the completion of a full design document for object reuse classification.* There are two research challenges related to this portion of our agenda. We are currently performing a set of structured interviews with software developers who use HPS to identify unique classificatory facets. Meanwhile, we are working to construct the elements of the analysis method that, given a workable classification scheme, will enable software developers to identify potentially reusable objects.

In this article, we have attempted to give the reader an appreciation of the kinds of measures which it is practical to derive from an automated analysis of an integrated CASE system. STRESS, the Seer Technologies Repository Evaluation Software Suite, enhances the ability of managers to control repository-based software development. It also makes it practical for us, as researchers, to perform data-intensive empirical analyses of software development processes. Software reuse, as this paper suggests, is of particular interest in this environment.

REFERENCES

1. Banker, R. D., Kauffman, R. J., Wright, C., and Zweig, D. "Automating Reuse and Output Metrics in an Object-based Computer Aided Software Engineering Environment." *IEEE Transactions on Software Engineering*, forthcoming.

2. Banker, R. D., Kauffman, R. J., and Zweig, D. "Repository Evaluation of Software Reuse." *IEEE Transactions on Software Engineering*, forthcoming.

3. Kemerer, C. F. "How the Learning Curve Affects CASE Tool Adoption", *IEEE Software*, volume 9, number 3, May 1992, pp. 23-28.

4. Prieto-Diaz, R., and Freeman, P. "Classifying Software for Reusability." *IEEE Software*, January 1987, pp. 6-16.

We wish to thank Mark Baric, Gene Bedell, Vivek Wadhwa, Tom Lewis and Gig Graham, for access to managers, developers and data related to software development activities at Seer Technologies and its clients. We are also indebted to Len Ehrlich, Michael Oara and Tom Robben for assistance with various aspects of this research program. We offer special thanks to Charles Wright, currently on assignment with Seer Technologies in Switzerland; his contributions to the automated function point analysis and software reuse analysis facilities that are described in this paper helped to make the work possible.