Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo

HOW REUSE INFLUENCES PRODUCTIVITY IN OBJECT-ORIENTED SYSTEMS

ALTHOUGH reuse is assumed to be especially valuable in building high-quality software, as well as in OO development, limited empirical evidence connects reuse with productivity and quality gains. The authors' eight-system study begins to define such benefits in an OO framework, most notably in terms of reduced defect density and rework, as well as in increased productivity.

THIS article presents the results of a study conducted at the University of Maryland in which we assessed the impact of reuse on quality and productivity in object-oriented (OO) systems. Reuse is assumed to be an effective strategy for building high-quality software. However, there is currently little empirical information about what to expect from reuse in terms of productivity and quality gains.

The study is one step toward a better understanding of the benefits of reuse in an OO framework in light of currently available technology. Data was collected for four months—September through December 1994—on the development of eight small (less than 15,000 source lines of code [KSLOC]) systems with equivalent functional requirements. All eight projects were developed using the Waterfall-style Software Engineering Life Cycle Model, an OO design method, and the C++ programming language. The study found significant benefits from reuse in terms of reduced defect density and rework as well as increased productivity. These results can also help software organizations assess new reuse technologies against a quantitative and objective baseline of comparison.

Software reuse can help produce quality software more quickly. Software reuse is the process of using existing software artifacts instead of building them from scratch [18]. Broadly speaking, the reuse process involves three steps:

- Selecting a reusable artifact
- Adapting it to the purpose of the application
- Integrating it into the software product under development

The major motivation for reusing software artifacts is to decrease software development costs and cycle time by reducing the time and human effort required to build software products. Some research [3, 11, 21] suggests that software quality can be improved by reusing quality software artifacts. Some work has also hypothesized that software reuse is an important factor in reducing maintenance costs because, when reusing quality objects, the time and effort required to maintain software products can be reduced [4, 19]. Thus, the reuse of software products, software processes, and other software artifacts is considered the technological key to enabling the software industry to achieve required levels of productivity and quality [7].

This article assesses the impact of product reuse on software quality and productivity in the context of OO systems. OO approaches are assumed to make reuse more efficient from both financial and technical perspectives. However, there is little empirical evidence that high efficiency is actually achieved with current technology. Therefore, what's needed is a better understanding of the potential benefits of OO for reuse—as well as current OO limitations. We view several quality attributes as dependent variables, including rework effort and number/density of defects found during the testing phases.

Validating the Hypotheses

Participants in our empirical study were the students of a graduate-level class offered by the Department of Computer Science at the University of Maryland. The class's objective was to teach OO software analysis and design. The students were not required to have previous experience or training in the application domain or in OO methods. All students had some experience with C or C++ programming and relational databases and therefore had the basic skills needed for the study.

To control for differences in skills and experience, the students were randomly grouped into eight teams of three students per team. To ensure the teams were comparable with respect to the ability of their members, the following two-step procedure (known as blocking [17]) was used to assign students to teams:

- Each student's level of experience was characterized. We used questionnaires and performed interviews. We asked the students about their previous working experience, their student status (part-time or full-time), their computer science degree (B.S., M.S., Ph.D.), their previous experiences with analysis/design methods, and their skill in various programming languages.
- Each of the eight most experienced students was randomly assigned to a different team. Students considered most experienced were computer science Ph.D. candidates who had already implemented large (less than or equal to 10 KSLOC) C or C++ programs and those with industrial experience of more than two years in C programming. None of the students had experience in OO software analysis and design methods. Similarly, each of the eight next most experienced students was randomly assigned to different groups; this random assigning was repeated for the remaining eight students.

Each team was asked to develop a management information system supporting the rental/return process of a hypothetical video rental business and the maintenance of customer and video databases. Such an application domain had the advantage of being easily comprehensible; therefore, we could make sure that system requirements could be easily interpreted by students regardless of their educational background.

The development process was performed according to a sequential software engineering lifecycle model derived from the Waterfall model and including the following phases: analysis, design, implementation, testing, and repair. A document was delivered

at the end of each phase—analysis document, design document, code, error report, and modified code. Analysis and design documents were checked to verify they matched the system requirements. Errors found in these first two phases were reported to the students. This verification and error checking maximized the chances that implementation would begin with a correct OO analysis/design. Acceptance testing was performed by an independent group. During the repair phase, the students were asked to correct their system based on the errors found by the independent test group.

The Object Modeling Technique (OMT), an OO analysis and design method, was used during the analysis and design phases [20]. The C++ programming language, the GNU software development environment, and OSF/MOTIF were used during implementation. Sun Microsystems Sparc workstations were used as the implementation platform—a development environment and technology representative of what is currently used in industry and academia. Our results are thus more likely to be generalizable to other development environments.

Rework seems to be lower in high-reuse categories, but there is no statistically significant evidence that faults are easier to detect and correct.

We provided the students with three libraries:

- **MotifApp.** This public-domain library includes C++ classes on top of OSF/MOTIF for manipulating windows, dialogs, and menus [22]. The MotifApp library provides a way to use the OSF/Motif widgets in an OO programming/design style.
- **GNU library.** This public-domain library is in the GNU C++ programming environment and contains functions for manipulation of strings, files, lists, and more.
- **C++ database library.** This library gives a C++ implementation of multi-indexed B-Trees.

We also provided a specific domain application library to make our study more representative of industrial conditions. This library implemented a graphical user interface (GUI) for insertion and removal of customer records and was implemented in such a way that the main resources of the OSF/Motif widgets and MotifApp library were used. Therefore, the library contained a small part of the implementation required for developing the rental system.

No special training was provided to teach the students how to use these libraries. However, the students received a tutorial describing how to implement OSF/Motif applications. In addition, a C++ programmer familiar with OSF/Motif applications was available to answer questions about the use of OSF/Motif widgets and the libraries. A hundred small programs exemplifying how to use OSF/Motif widgets were also provided. In addition, the code sources and the complete documentation of the libraries were provided. Finally, it should be noted that the students were not required to use the libraries and that, depending on the particular design they adopted, different reuse choices were expected.

To define the metrics to be collected during the experiment, we used the Goal/Question/Metric (GQM) paradigm [5, 7]. The study's goal was to analyze reuse in an OO software development process for evaluation with respect to rework effort, defect density, and productivity from an organizational point of view. In other words, our objective was to assess the following assumptions in the context of OO systems developed under currently available technology:

- A high reuse rate results in a lower likelihood of defects.
- A high reuse rate results in lower rework effort, that is, less effort to repair software products.
- A high reuse rate results in higher productivity.

According to the GQM paradigm, we had to define a set of questions pertinent to the defined experimental goal and a set of metrics allowing us to devise answers to these questions. We do not present the complete GQM here, only the metrics we derived. However, the metrics described in the next section were derived by following the GQM methodology [6].

Independent and Dependent Variables

Here we define the study's independent variables (e.g., size, amount of reuse) and dependent variables (e.g., productivity, defect density). We intend to make the underlying assumptions and models clear, so a precise terminology is used in the rest of the article. A thorough and formal discussion of these issues can be found in [10].

The size of a system S is a function *Size(S)* characterized by several properties, including the following:

- Size cannot be negative (property Size.1).
- We expect size to be null when a system does not contain any component (property Size.2).
- More important, when components do not have elements in common, we expect *Size* to be additive (property Size.3).

From these simple properties, other properties can be derived, as discussed in [10].

Let us assume an operator called *Components,* which, when applied to a system S, gives the distinct components of S, so that:

Components(S) = $\{C_1, ..., C_n\}$, such that if $C_i = C_j$ then i=j, where i, j=1,...,n.

The size of a system S is given by the following function:

$$Size(S) = \sum_{c \in Components(S)} Size(c)$$

where Size(c) can be defined as, for instance, the number of source lines of code of the component c. However, as discussed later, measuring size in the context of reuse raises difficult measurement issues related to such OO mechanisms as inheritance and aggregation of classes [20].

The amount of reused code in a system S is a function *Reuse(S),* also characterized by the properties Size.1–Size.3; that is, *Reuse(S)* is an instance of a size metric. Therefore, *Reuse* cannot be negative (property Size.1), and we expect it to be null when a system does not contain any reused element (property Size.2). When reused components do not have reused elements in common, we expect *Reuse* to be additive (property Size.3).

The way we define *Reuse(S)* must take into account specific OO concepts, such as classes and inheritance. For instance, consider a class C, which is included in a system S. There are five cases:

1. Class C belongs to the library LC. In this case we have verbatim reuse, that is, an existing class is included in the system S without being modified. Therefore:

   $Reuse(C) = Size(C)$

   As we are dealing with an OO language allowing inheritance, all ancestors of C, as well as all classes aggregated by C, also have to be included in S. As all C ancestors and all classes aggregated by C also belong to the library, including a library class may trigger an apparent large amount of verbatim reuse.

2. Class C is a new class created by specializing, through inheritance, a library class LC. This case is a variation of the first case, that is, the class LC and all its ancestors and subclasses (aggregated classes) will be included in S and will be dealt with in a way similar to verbatim reuse.
3. Class C is a new class that aggregates a library class LC. This case is also a variation of the first case, that is, the class LC and all its ancestors and subclasses will be included in S and will be considered in a way similar to verbatim reuse.
4. Class C has been created by changing the existing class EC. Reuse can be estimated as:

   $Reuse(C) = (1 - \%Change) \times Size(C)$

   where %Change represents the percentage of C added to or modified from EC.

   However, the percentage of change is difficult to obtain. As a simplification, we asked the developers to tell us if more or less than 25% of a component had been changed. In the former case, the class was labeled as extensively modified; in the latter case, the class was labeled as slightly modified. Therefore, reuse rates were computed based on the following approximations:

   - Extensively modified: $Reuse(C) = 0$
   - Slightly modified: $Reuse(C) = Size(C)$

   We show later that slightly modified and verbatim reused components are quite similar from the point of view of defect density and rework. Thus, the approximation appears to be reasonable.

5. Class C was created from scratch. In this case, the amount of reuse of the class C is 0:

   $Reuse(C) = 0$

   Now assume a function called *Classes*, which when applied to a system S, yields all classes of the system S, so that:

   $Classes(S) = \{C_1, ..., C_n\}$, such that if $C_i=C_j$, then $i=j$, where $i,j=1,...,n$.

   Reuse of a system S is given by the following function:

   $$Reuse(S) = \sum_{c \in Classes(S)} Reuse(c)$$

We are also particularly interested in knowing the reuse rate in a particular system. Reuse rate is measured by the following function:

$$ReuseRate(S) = Reuse(S)/Size(S)$$

This metric has the property of being normalized: $0 \leq ReuseRate(S) \leq 1$.

The first three cases show that the size measures of systems can be artificially inflated. Only a more detailed static analysis of the code would permit more precise size measurement by distinguishing what is actually used from what is inherited. This issue is addressed when defining measures of productivity and defect density.

Here we are interested in estimating the effort breakdown for development phases and for error correction:

- Person-hours per development activity, including:
  Analysis. The number of hours spent understanding the concepts embedded in the system before any actual design work, including requirements definition and requirements analysis, as well as analysis of changes made to requirements or specifications, regardless of where in the life cycle they occur.
  Design. The number of hours spent performing design activities, such as high-level partitioning of

the problem, drawing design diagrams, specifying components, writing class definitions, and defining object interactions. The time spent reviewing design material, such as doing walk-throughs and studying the current system design, was also taken into account.

Implementation. The number of hours spent writing code and testing individual system components, including: person-hours per error (referred to as rework), such as number of hours spent isolating an error and correcting it. We are also interested in rework efficiency, that is, how easily modifiable a class or a system is. To measure such an attribute, we normalize rework effort by the size of classes and the number of faults, or changes, respectively.

Here we are interested in measuring the productivity of each team. The measure used was the amount of code delivered by each project vs. the effort to develop such code, so:

Productivity(S) = Size(S)/DE(S)

where, in the study:

- Size(S) is first operationally defined as the number of lines of code delivered in the system S. Other size measures, such as function points, could have been used, but lines of code fulfilled our requirements and could be collected easily. More important, we were looking at the relative sizes of systems addressing similar requirements and therefore of similar functionality. However, because of the effect of inheritance on size measurement, we also measured size by excluding verbatim reused classes. This exclusion is not fully satisfactory because it underestimates the size of systems with a large amount of verbatim reuse classes. Nevertheless, it provides an additional insight on productivity complementary to our first measure. In addition, since all systems are supposed to be functionally equivalent, we also measured productivity by assuming that systems all have an equivalent size; therefore, effort was assumed to be inversely proportional to productivity. Again, this relationship is an approximation and is another interesting way to look at productivity.
- DE(S) (development effort) is defined as the total number of hours a group spent analyzing, designing, implementing, and repairing the system S.

Here we analyze the number and density of defects for each system component. We use the term *defect* generically to refer to either an error or a fault. Errors and faults are two pertinent ways to count defects, and both were considered in the study. Errors are defects in the human thought process made while trying to understand given information, to solve problems, or to use methods and tools. Faults are concrete manifestations of errors in the software. One error may cause several faults, and various errors may cause identical faults. Density is defined as:

Density(S) = #Defects(S)/Size(S),

where #Defects(S) is defined as the total number of defects detected in the system S during test phases.

In the study, an error is assumed to be represented by a single error report form filled out by the independent tester group; a fault is represented by a physical change to a component, that is, in this particular context, a C++ class. Error density is first operationally defined as the number of errors found in a system over the number of KSLOC contained in the system. As for productivity, and for the same reasons, we also used a size measure excluding verbatim reused classes. Again we assumed that system sizes are roughly equivalent. In this case, defect counts were assumed to capture defect density.

Now assume that, in order to correct error E1, two classes—C1 and C2—have been modified, whereas, in order to correct E2, only class C2 was modified. In this case, the *fault* density of S is three faults per KSLOC.

To apportion errors to specific classes, we have to account for the fact that one error could result in changes to several classes. In this case, we follow a procedure illustrated by the following example:

- The error weight affecting C1 will be equal to 0.5 because two classes were modified to correct E1.
- The error weight of C2 will be equal to 1.5 because two classes were modified to correct E1, and only C2 was modified to correct E2.

This procedure is formally represented as:

$$| \mathrm{ErrorWeight}(C_i) | = \sum_{E_{ij} \in \{E_{ij} \dots E_{in}\}} | \mathrm{Classes_affected}(E_{ij}) |$$

where:

- $|$ ErrorWeight(C_i) $|$ is the error weight associated with the class C_i;
- $\{E_{ij} \dots E_{in}\}$ is the set of errors in which the class C_i was affected; and
- $|$ Classes_affected(E_{ij}) $|$ is the number of classes affected by the error E_{ij}.

We used the approach in [5], which proposes using forms for collecting data and gives guidelines for checking the accuracy of the information gathered. We used three different types of forms tailored from those used by the Software Engineering Laboratory [5]:

- **Personnel Resource Form.** This form is used to gather information about the amount of time the software engineers spent on each software development phase.

- **Component Origination Form**. This form is used to record information characterizing each component in the project under development at the time it gets into configuration management. This form is also used to capture whether the component has been developed from scratch or from a reused component. In the latter case, we collected the amount of modification—none, small, or large—needed to meet the system requirements and design, as well as the name of the reused component. By small/large, we mean less/more than 25% of the original code has been modified.
- **Error Report Form.** This form was used to gather data about (1) the errors found during the testing phase, (2) the components changed to correct such errors, and (3) the effort expended in correcting it. The last item includes:
  – Determining precisely what change was needed
  – Understanding the change or finding the cause of the error
  – Locating the point where the change was to be made
  – Determining that all effects of the change were accounted for
  – Implementing the correction, including design changes, code modifications, and regression testing

Validity

The study's validity can be analyzed from two perspectives: internal (What threats to the conclusions can we draw from the study?) and external (How generalizable are these results?) [17]. With respect to internal validity, we can say that subjects were classified according to their ability and assigned randomly to form "equivalent" teams. Therefore, we are less likely to obtain biased results due to differences in ability across teams. However, performing a formal controlled experiment would have required assignment of random levels of reuse to projects and classes—not feasible in practice. And because of this inability to assign random levels of reuse, reuse rates might be associated with other factors.

Concerning external validity, we can say that even though students are not industrial programmers, we have trained them thoroughly and we used an application domain intuitive enough to avoid misunderstandings when interpreting the requirements. In addition, we used a development environment representative of what is available in industry for OO software development. Another possible threat to external validity is that our systems were relatively small, so their conceptual complexity may be limited when compared to large software development applications. However, the inherent limitation of such empirical studies can't be avoided.

Software Product Reuse and Software Quality

We analyzed the impact of code reuse on software quality, investigating two aspects of quality: defect density and rework. We present the results in two ways:

- Assessing the differences in quality across reuse categories—new, extensively modified, slightly modified, verbatim
- Computing an approximate project reuse rate and assessing its statistical association with project quality

In the first form of analysis, projects are considered as separate entities; in the second, trends across projects are analyzed in a way that assumes the projects are comparable. In addition, the first type helps us justify the definition of the reuse metric we used for the study.

Note that during the analysis we did not distinguish between "horizontal" and "vertical" reuse; that is, the code reused from the generic libraries and the code reused from the domain-specific libraries have been combined. Even though comparing the benefits of these two kinds of software product reuse would be interesting, it is beyond the scope of this article.

The study found significant benefits from reuse in terms of reduced defect density and rework as well as increased productivity.

Reuse Vs. Defect Density

The first analysis compared reused and newly created code from the perspective of defect density. We looked at defects according to two definitions: errors and faults. At the class level, we used a simple measure of size—lines of code—but other size measures could have been used for the same purpose. However, since we were comparing systems developed based on identical requirements and of similar functionalities, we think this simple and convenient size measure is at least precise as a relative measure between projects. At the system level, three different measures of defect densities were investigated.

We first examined the relationship between classes and defect density to see if reused classes are less prone to defects. In addition, we used this analysis as an opportunity to evaluate the value of our ordinal reuse measure and assess its effect on defect density.

Table 1 shows the error and fault densities (errors and faults per thousand lines of code) observed in each of the four categories of class origin. Apparently, fewer defects were found in reused code. For example, error density was found to be only 0.125 in the code reused verbatim, 1.50 in the slightly modified code, 4.89 in the extensively modified code, and 6.11 in the newly developed code.

However, these differences should be assessed statistically; the significance of these trends should be calculated. To perform this statistical analysis, defect densities were computed for each project and each reuse category; the results were combined to derive defect densities for each project's subset of classes belonging to each reuse category. When comparing reuse categories, we are in fact comparing sets of defect density values, each corresponding to a given project. Each observation in each reuse category therefore matches one observation in each of the other reuse categories, since they correspond to the same system and have been developed by an identical team. In addition, classes across reuse categories have similar complexity and comparable functionalities. Conceptually, it is almost like looking at the characteristics of identical sets of classes developed with different reuse rates.

The way we measure reuse and size needs to be refined to obtain more accurate measurement of what is actually used by the system as opposed to what is inherited.

Considering that eight systems were developed for the study, eight independent observations are available at the system level. The data set is rather small; consequently, we adopted a data analysis strategy following several steps:

- First, we used a nonparametric test (Wilcoxon matched-pairs signed rank test or Wilcoxon T test [12]) to determine whether significant differences could be observed between reused, modified, and newly developed classes. The rationale underlying this test is straightforward. We are comparing a set of pairs of scores (in this case, defect densities). Suppose the score for the first member of the pair is DD_1, and the score for the second member of the pair is DD_2. For each pair, we calculate the difference between the scores as $DD_2 - DD_1$. The null hypothesis we wish to test is that there is no difference between pairs of scores for the population from which the sample of pairs is drawn; that is, that there is no difference with respect to defect density between reuse categories. If this is true, we would expect similar numbers of negative and positive differences and similar magnitudes of differences. Therefore, the Wilcoxon T test takes into account both the direction and the magnitude of differences between scores, or defect densities, to determine whether the null hypothesis is reasonable. Thus, by using this test, we can obtain a statistical comparison of any project characteristic, such as defect density, across class reuse categories. Such a test does not assume all projects are comparable, does not require more than five observations per reuse category, and is robust to outliers, or extreme differences in scores between pairs. All these properties are important in our study.
- Once it has been determined that there is a significant difference between reuse categories, our goal is to quantify to the best extent possible the impact of reuse on defect density. To do this, we perform a linear least-squares regression between reuse rate and defect density. It might be argued that the number of data points we are working with is too small to allow such an analysis. However, there is common agreement that the number of independent observations per explanatory variable could be as low as five [14].

Another analysis strategy would have been to work at the class level by, say, comparing defect densities of classes across reuse categories, but this strategy presented problems:

- Some of the projects included a large percentage of all reused classes. Therefore, it would have been difficult to determine whether the observed trends could be attributed to skill differences between teams or to reuse.
- Our defect density and productivity measures can be considered suitable at the system level but are too rough at the class level.

Table 1. Error/fault densities and rework in each class origin category for all projects

Class Origin	No. of Comp.	No. of SLOC	No. Faults	Fault Density	No. of Errors	Error Density	Rework
New	177	25,642	247	9.63	157	6.11	336.35
Extensively Modified	79	15,165	93	6.13	74	4.89	160.04
Slightly Modified	45	6,685	11	1.57	10	1.50	22.5
Reused Verbatim	92	16,015	6	0.37	2	0.12	3
All Classes	**393**	**63,537**	**356**	**4.88**	**243**	**3.82**	**521.89**

We first looked at *fault* densities. Figure 1 shows the distribution and mean of project fault densities across reuse categories. Each diamond schematically represents the mean for each class reuse category. The line across each diamond represents the category mean. The height of each diamond is proportional to the 95% confidence interval for each category, and its width is proportional to the cate-

Figure 1. Distribution and mean of project fault densities across reuse categories

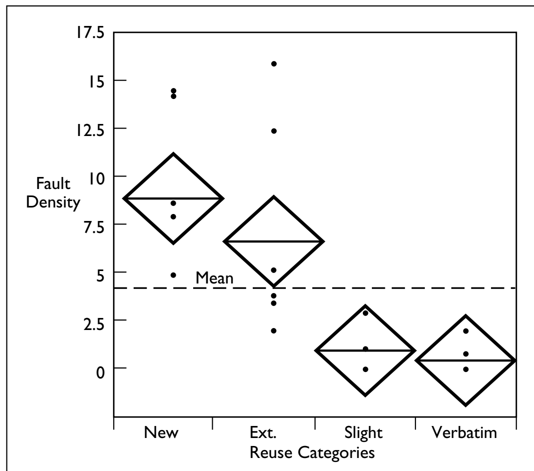
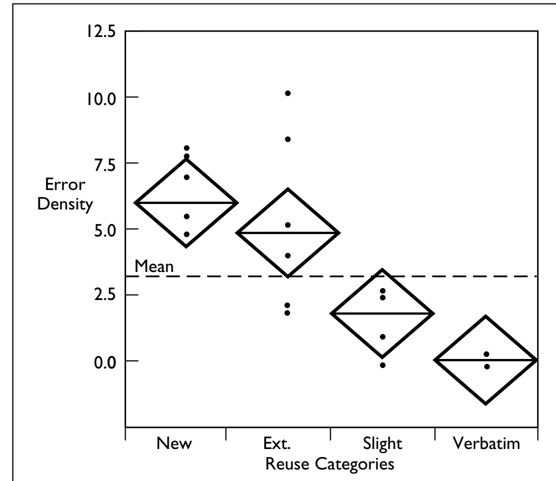


Figure 2. Distribution and mean of project error density across reuse categories



gory sample size. Recall that we want to know whether reuse reduces defect-proneness; in other words, the null hypothesis is: Reused and nonreused classes are, on average, of "similar defect-proneness." However, to gain confidence in the results, we have to check that the internal structure of the classes (in each reuse category) did not play a role in the outcome. We ran an analysis using various code metrics (e.g., cyclomatic complexity, nesting level, and function calls) and determined that the distributions across reuse categories were not statistically different. (These measures were extracted using the Amadeus tool [2].)

Table 2 shows the paired statistical comparisons of fault densities between reuse categories. We assumed significance at the 0.05 α-level; that is, if the p-value is greater than 0.05, we assume there is no observable difference. Recall that p-values are estimates of the probabilities that differences between reuse categories (in this case, in terms of project fault densities) are due to chance. According to these results, there is no support for the fact that there is an observable difference between verbatim reuse and slightly modified code, or between extensively modified and new code. This means that from the perspective of fault density, extensively modified code does not bring much benefit and that slightly modified code is nearly as good as code reused verbatim.

We used the same approach to obtain a statistical comparison of class error density per class category. Error density is more complicated to compute with respect to reuse categories, as one error may trigger changes in several classes from different categories. We calculated the error weight per class; then, for each class category, we computed the sum of the classes' error weights for each project and divided it by the sum of the sizes of those classes.

This assumption is not so strong, since (1) in general, each error generates only one fault, and (2) when an error generates many faults, in most cases all classes affected belonged to the same reuse category. Therefore, all errors were considered with equal weight. The distributions are shown in Figure 2, and the Wilcoxon T-test results are shown in Table 3.

Again there is no observable difference between verbatim-reused and slightly modified classes (even though the significance improved compared to fault analysis results, it is still greater than 0.05) or between extensively modified and newly created classes. This lack of difference means that from the perspective of error density, extensively modified code does not bring much benefit and slightly modified code is as good as code reused verbatim. These results confirm the results we obtained using fault density as a quality measure.

We also wanted to verify the hypothesis that the higher the project reuse rate, the lower the number of project errors. For the sake of simplification, only

Table 2. Levels of significance (fault density per reuse category)

p-values	Slight	Ext.	New
Verbatim	0.46	0.012	0.012
Slight		0.012	0.012
Ext.			0.26

Table 3. Levels of significance (error density per reuse category)

p-values	Slight	Ext.	New
Verbatim	0.08	0.012	0.012
Slight		0.0136	0.025
Ext.			0.26

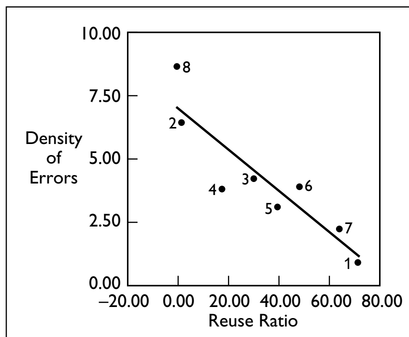
Project	No. of SLOC	No. of Errors	Reuse Rate	Error Density (with verbatim reuse code)	Error Density (without verbatim reuse code)	Rework
1	13,981	24	47.29	1.72	3.48	51
2	5,068	33	2.23	6.51	6.60	71
3	9,735	42	31.44	4.31	5.42	92
4	8,543	33	18.08	3.86	3.86	72
5	8,173	26	40.05	3.18	4.78	59
6	6,368	25	48.67	3.93	6.15	51
7	6,571	15	64.01	2.28	2.96	31
8	5,068	44	0.00	8.68	9.36	93

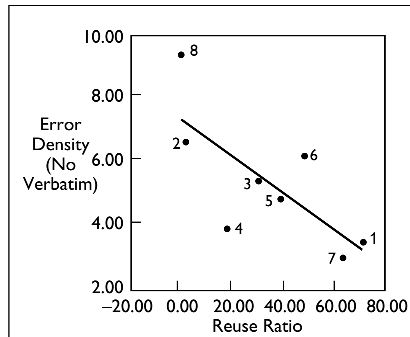
Table 4. Overview of the projects' data

verbatim-reused and slightly modified classes were considered "reused classes" for computing the reuse rate per project. This approximation was to some extent justified by the results discussed earlier showing extremely different trends for slightly and extensively modified classes. We see this analysis as complementary to the earlier analysis, since the determination of the relationship between reuse rate and error density allows us to quantify the impact of reuse. However, the drawbacks of this new analysis are that all projects were assumed to be comparable and that the results could easily have been biased by outliers. Table 4 provides an overview of the projects' data, including number of lines of code delivered at the end of the implementation phase, reuse rate per project, error density (including and excluding verbatim reuse), and rework hours.

The average rate of reuse is approximately 31%, with a maximum of 64%. Based on Figure 3a, there appears to be a strong linear relationship between reuse rate and project error density when verbatim reuse is included in system size. This relationship is statistically significant (p-value = 0.0051 when performing an F-test) and shows a high coefficient of determination (R^2= 0.755). The estimated intercept and slope are 7.02 and -0.086, respectively. That means that when there is no reuse, error density should be expected to be around 7, and each additional 10 percentage points in the reuse rate decrease this density by nearly 1 (the estimate is 0.86) within the range covered by the data set (we limit ourselves to interpolation). No outlier seems to be the cause of a spurious correlation; therefore, this result should be meaningful (see Figure 3a).

Figures 3b and 3c show the relationships between reuse rate and, respectively, error density without verbatim reused classes (R^2= 0.54 and p-value = 0.04) and number of errors (R^2= 0.66 and p-value = 0.01). In both cases, a significant negative relationship can be observed, confirming our interpretation of the relationship identified in Figure 3a. In other words, we obtained consistent results using three different measures of error density as a dependent variable. Since these measures are based on very different assumptions, we are pretty confident in saying that reuse has a strong and positive impact on error-proneness.





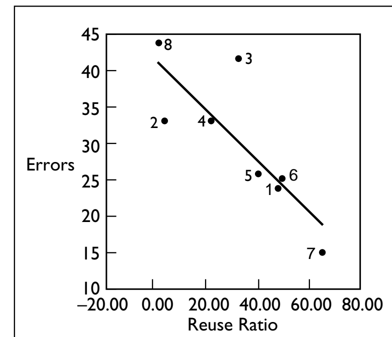


Figure 3a. Linear relationship between error density and reuse rate
Figure 3b. Linear relationship between error density without verbatim reuse code and reuse rate
Figure 3c. Linear relationship between a project's number of errors and its reuse rate

These results support the assumption that reuse in OO software development yields a lower defect density. For example, the participants in Project 8 decided to implement everything from scratch; reuse did not have an impact on error density in their case. All the participants in the other projects performed better, and their error densities appear

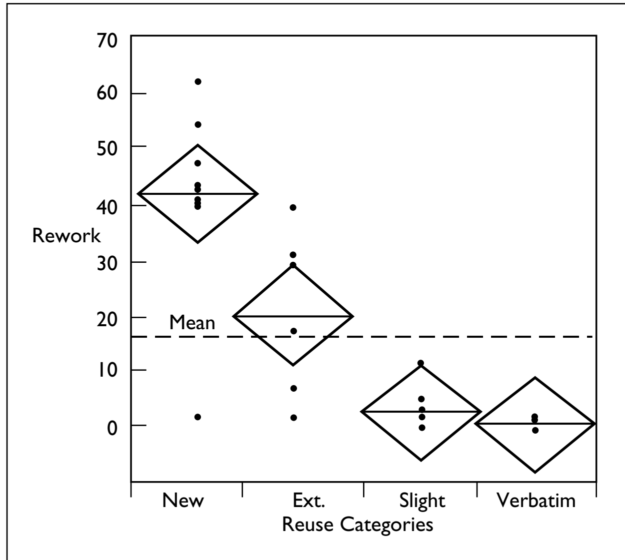


Figure 4. Distribution and mean of project rework per reuse category

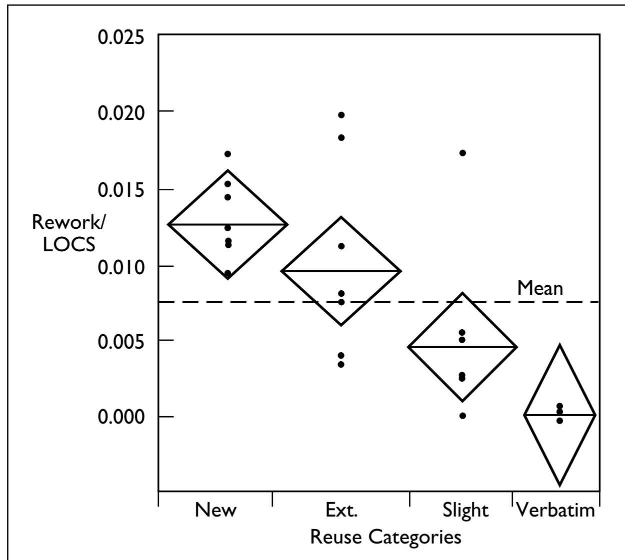


Figure 5. Distribution and mean of project rework density per reuse category

to decrease linearly with the reuse rate. This is strong evidence that reuse helped improve quality across the covered reuse rate range, that is, 0% to 64%.

In [16], rework is identified as a major cost factor in software development. Rework on the average accounts for more than 50% of the effort for large projects [16]. Reuse of previously developed, reviewed, and tested classes could result in easy-to-maintain classes and consequently should decrease the rework effort. Here, we first compare rework effort on reused and newly created classes. We then check whether the total amount of reuse per project is related to a reduction in the project rework effort.

Reuse Vs. Rework

We are interested in seeing whether the effort needed to repair reused classes is lower than the effort needed to repair classes created from scratch or extensively modified. We looked at three different measures to answer this question:

1. Total amount of rework in each class reuse category
2. Rework normalized by the size of the classes belonging to each reuse category
3. Rework normalized by the number of faults detected in the class of each reuse category

Distributions and means are shown in Figures 4, 5, and 6.

These metrics allowed us to look at rework from various perspectives:

- Capturing the total cost of rework, expected to be somewhat associated with the size of the classes and the number of faults, or changes, in each reuse category.
- Allowing us to look at rework without considering the relative amount of code in each reuse category, which gives us a more accurate insight into the relative cost of debugging and perfecting code in each reuse category.
- Allowing us to look at the expected difficulty of repairing a single fault across the various reuse categories, which gives us a more accurate insight into the modifiability of classes independent of their fault-proneness.

Before any thorough statistical analysis, a look at the distributions seems to indicate measures 1 and 2 are significantly different across reuse categories. To test the significance of this difference, we ran a Wilcoxon T test [12]. Instead of using defect density per reuse category as scores, we used total amount of rework

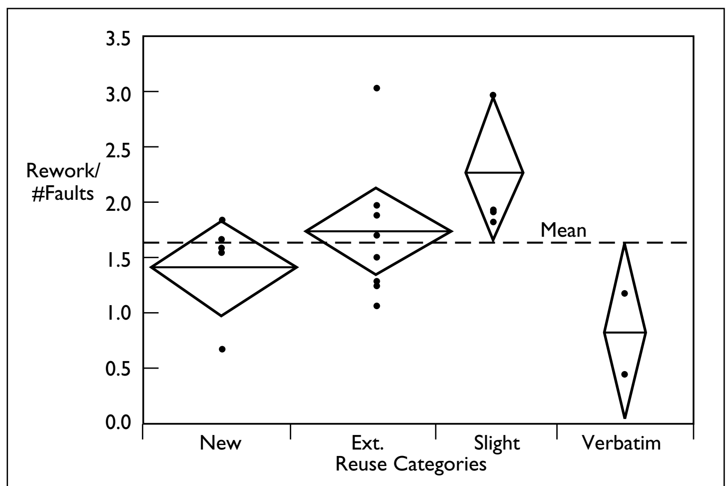


Figure 6. Distribution and mean of project rework difficulty per reuse category

per reuse category. The results for the first metric are shown in Table 5.

Based on Table 5, we can conclude that reuse reduces the amount of rework, even when the code is extensively modified. Again there is no observable difference between the verbatim-reused and slightly modified classes. These results show that from the perspective of total rework, extensively modified code might still bring benefits and, once again, slightly modified code is nearly as good as code reused verbatim.

Table 5. Rework per reuse category

p-values	Slight	Ext.	New
Verbatim	0.138	0.018	0.012
Slight		0.025	0.017
Ext.			0.05

As for defect density and rework, we used the Wilcoxon T test to analyze the variations of the second metric, that is, rework normalized by the size of the classes belonging to each reuse category (referred to as rework density) across reuse categories. Results are presented in Table 6.

Based on Table 6, we conclude that reuse reduces rework density except when the code is extensively modified. There is no observable difference between the verbatim-reused and slightly modified classes. These results show that from the perspective of rework density, reuse brings benefits and slightly modified code is nearly as good as code reused verbatim.

Some projects had no faults in their verbatim or slightly reused classes. The number of data points in these categories thus became too small for applying a Wilcoxon T test to the third metric. In such cases, we could look only at the difference between new and extensively modified classes. No significant difference could be observed in this case. As a last attempt to look at change difficulty (the third metric), we performed an analysis at the class level, where rework effort per class was normalized by the number of faults detected and corrected in these classes. No significant differences in distribution could be observed across reuse categories. If these results were confirmed by further studies, it would mean that differences in rework effort across reuse categories would be mainly due to differences in fault-proneness and not to differences in ease of modification.

To conclude, rework seems to be lower in high-reuse categories, but there is no statistically significant evidence that faults are easier to detect and correct.

To complement these results, we would like to verify whether larger project reuse rate is associated with lower project total rework effort. Even though the number of data points available is small, we can observe a strong linear relationship between rework and project reuse rates that is statistically significant ($p = 0.015$ on F-test), with a coefficient of determination R^2 of 0.65. The estimated intercept and slope are 88.52 and −0.748, respectively. That means that, where there is no reuse, rework effort should be expected to be around 88 person-hours for each project and that for each additional 10 percentage points in reuse rate (within the reuse rate interval covered by our data set) rework effort will decrease by nearly 7.5 person-hours. These results are, of course, specific to the system requirements implemented in the study, but they could be generalized as follows: Each additional 10 percentage points in reuse rate, within the reuse rate interval covered by our data set, decreases rework by nearly 8.5%. Again, no outlier seems to be causing any spurious correlation (see Figure 7).

Table 6. Rework/SLOC per reuse category

p-values	Slight	Ext.	New
Verbatim	0.144	0.043	0.043
Slight		0.028	0.042
Ext.			0.16

Table 7. Overview of the projects' data collected after the errors have been fixed

Project	SLOC Delivered	Reused	Reuse Rate	Effort	Productivity	Productivity (without verbatim reused code)
1	14,222	6,611	46.48	155	91.75	47.55
2	5,105	113	2.21	280	18.23	17.70
3	11,687	3,061	26.19	365	32.01	18.28
4	10,390	1,545	14.87	303	34.3	23.10
5	8,173	3,273	40.04	159	51.4	30.82
6	8,216	3,099	37.71	264	31.12	12.38
7	9,736	4,206	43.2	140	69.54	16.89
8	5,255	0	0	264	19.9	19.20

To better capture the concept of rework, it would be better to look at rework normalized by the size of the changes that occurred during the repair phase. Unfortunately, we could not capture this information accurately with the data collection procedures we had in place. As a rough approximation, we looked at rework normalized by the number of faults. However, no significant differences were observed between reuse categories. This result was confirmed when we attempted to investigate rework normalized by the number of faults at the class level.

In conclusion, the results support the assumption that reuse in OO software development results in lower rework effort.

Software Product Reuse and Software Productivity

Reuse has been advocated as a means of reducing development cost. For example, in [9], reuse of classes is identified as one of the most attractive strategies for improving productivity. As productivity is often considered an exponential function of software size, a reduction in the amount of software to be created could provide a dramatic savings in development costs [8]. The question now is, to what extent does reuse improve productivity—despite change and integration costs?

Table 7 shows, for each project analyzed:

- Number of lines of code delivered at the end of the lifecycle
- Number of lines of code reused (verbatim reused and slightly modified classes)
- Reuse rate
- Effort
- Productivity, including verbatim reused code
- Productivity, excluding verbatim reused code

Note that the data in the reuse rate and SLOC delivered column are different from the data in Table 4. This difference stems from the fact that Table 8 presents the results at the end of the lifecycle, that is, after the errors have been fixed, whereas Table 4 presents the data collected at the end of the implementation phase.

BASED on data in Figure 8a, we can conclude that there is also a strong linear relationship ($R^2 = 0.666$), which is statistically significant (p-value = 0.013 ON an F-test) between productivity (including verbatim reuse) and reuse rate. The estimated intercept and slope are 14.04 and 1.11, respectively. When there is no reuse, productivity should be expected to be around 14 SLOC per hour and each additional 10 percentage points in the reuse rate should increase productivity by 11 SLOC per hour. Figures 8b and 8c show the relationships between reuse rate and, respectively, productivity without verbatim reused classes ($R^2 = 0.45$ and p-value= 0.067) and effort ($R^2 = 0.38$ and p-value = 0.099). In Figure 8b, a weaker positive relationship (significant at the a = 0.1 level) can be observed, confirming our interpretation that productivity has improved. Figure 8c shows a weak negative trend (expected to be negative because the dependent variable is effort), also supporting our claim about productivity improvement. However, the latter figure is graphically and statistically not as clear as the other two figures, due to the third observation, which is clearly an outlier.

To explain outliers on these scatterplots, we performed some qualitative analysis of the process used, the teams involved, and the design strategies adopted in each project. For example, the team in Project 3 had no previous experience with respect to GUIs, and learning the basics was perceived as a significant effort. Similarly, Project 6 appears to have had lower productivity than expected in Figures 8a, 8b, and 8c when considering reuse rate. Lower productivity was explained by the particularly sophisticated GUI this group designed. In the context of the requirements we provided to the students, the GUI could be considered gold-plating.

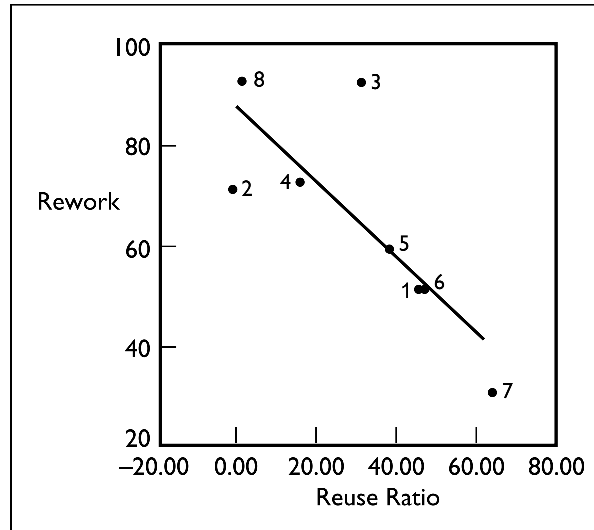
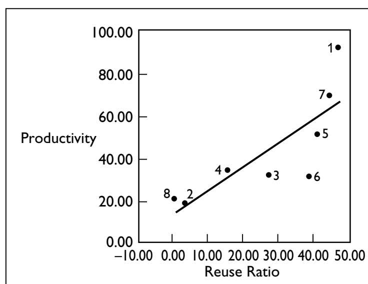
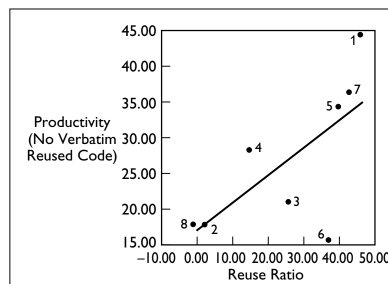


Figure 7. Linear relationship between rework and reuse rate

Conclusion

This article offers significant results showing the strong impact of reuse on product productivity and,





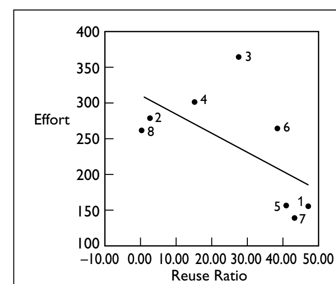


Figure 8a. Linear relationship between productivity and reuse rate
Figure 8b. Linear relationship between productivity (without verbatim reuse code) and reuse rate
Figure 8c. Linear relationship between development effort and reuse rate

especially, on product quality, or defect density and rework density, in the context of OO systems. In addition, these results were obtained in a common and representative OO development environment using standard OO technology. Such results can be used as rough estimates by managers and as a baseline of comparison in future reuse studies for the purpose of evaluating reuse processes and technologies.

This study can and must be replicated in industry and academia. In industry, replicating this study can, for example, help managers decide whether it is worth investing in particular OO technologies to improve software quality and productivity. In academia, replicating this study can help test OO methods or compare the advantages of such methods against those of traditional development methods. In any case, replication is necessary to confirm the results we obtained and refine the models we built.

Future work includes refinement of the information collected during the repair phase with regard to the size and complexity of the changes. This would allow us to better estimate the impact of reuse on rework. However, it is likely to require better automation of the change data and therefore the design of tools for monitoring the changes to code and design documents. In addition, the way we measure reuse and size needs to be refined to obtain more accurate measurement of what is actually used by the system as opposed to what is inherited. Thus, we should be able to measure productivity and defect density more precisely.

We also intend, in future replications of this experiment, to assess independently the impact of horizontal (non-domain-specific) and vertical (domain-specific) software reuse on software quality and productivity. We will compare the advantages and drawbacks of using these two types of software libraries. Finally, it would be interesting to refine our comparison of the internal class characteristics across reuse categories by using more specific OO metrics [1]. We need to better characterize the impact of reuse on system size, complexity, coupling, and cohesion. ◘

Acknowledgments
We want to thank Gianluigi Caldiera for helping us teach the OMT method; Khaled El Emam, Jyrki Kontio, Carolyn Seaman, and Barbara Swain for their suggestions, which helped improve both the content and the form of this article; and the students of the University of Maryland for participating in this study.

This work was supported, in part, by NASA Grant NSG-5123, NSF Grant 01-5-24845, Fraunhofer Gesellschaft, UMIACS, and Westinghouse Corp.

References

1. Abreu, F.B., and Carapuça, R. Candidate metrics for object-oriented software within a taxonomy framework. *J. Syst. Software 26*, 1 (1994), 87–96.
2. Amadeus Software Research, Inc. *Getting Started with Amadeus.* Amadeus Measurement System, Irvine, Calif., 1994.
3. Agresti, W.W., and McGarry, F. The Minnowbrook workshop on software reuse: A summary report. In *Software Reuse: Emerging Technology*, W. Tracz, Ed., IEEE Press, 1987.
4. Basili, V. Viewing maintenance as reuse-oriented software development. *IEEE Software 7*, 1 (Jan. 1990), 19–25.
5. Basili, V., and Weiss, D.M. A methodology for collecting valid software engineering data. *IEEE Trans. Software Eng. 10*, 6 (Nov. 1984) 728–738.
6. Basili, V., and Rombach, H.D. Support for comprehensive reuse. IEEE Software Engineering Journal (Sept. 1991), 303–316.
7. Basili, V., and Rombach, H.D. The TAME project: Towards improvement-oriented software environments. *IEEE Trans. on Software Eng. J. 14*, 6 (June 1988), 758–773.
8. Boehm, B.W. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, N.J., 1981.
9. Boehm, B.W., and Papaccio, P.N. Understanding and controlling software costs. *IEEE Trans. Software Eng. 14*, 10 (Oct. 1988), 1462–1476.
10. Briand, L., Morasca, S., and Basili, V. Property-based software engineering measurement. *IEEE Trans. Software Eng. 22*, 1 (Jan. 1996), 68–86.
11. Brooks, F.P. No silver bullet: Essence and accidents of software engineering. *Computer 20*, 4 (Apr. 1987).
12. Devore, J. *Probability and Statistics for Engineering and Sciences.* Brooks Cole Publishing Co. 1991.
13. Fenton, N.E. *Software Metrics: A Rigorous Approach.* Chapman & Hall, London, 1991.
14. Ferguson, G., and Takane, Y. *Statistical Analysis in Psychology and Education.* McGraw-Hill, New York, 1989.
15. Heller, G., Valett, J., and Wild, M. Data Collection Procedure for the Software Engineering Laboratory (SEL) Database. SEL Series, SEL-92-002. NASA/GSFC, Greenbelt, Md., 1992.
16. Jones, T.C. *Programming Productivity.* McGraw-Hill, New York, 1986.
17. Judd, C.M., Smith, E.R., and Kidder, L.H. *Research Methods in Social Relations.* Harcourt Brace Jovanovich College Publishers, 1991.
18. Krueger, C.W. Software reuse. *ACM Comput. Surv. 24*, 2 (1992), 131–183.
19. Rombach, H.D. Software reuse: A key to the maintenance problem. *Inf. Software Technol. J. 33*, 1 (Jan./Feb. 1991), 86–92.
20. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, N.J., 1991.
21. Thomas, W., Delis, A., and Basili, V. An evaluation of Ada source code reuse. In *Proceedings of the Ada-Europe International Conference* (Zandvoort, The Netherlands, June 1992).
22. Young, D.A. *Object-Oriented Programming with C++ and OSF/MOTIF.* Prentice-Hall, Englewood Cliffs, N.J., 1992.

VICTOR R. BASILI is a professor in the University of Maryland's Institute for Advanced Computer Studies and Computer Science Department. He can be reached at basili@cs.umd.edu.

LIONEL C. BRIAND is head of the quality and process engineering department at the Fraunhofer-Institute for Experimental Software Engineering in Germany. He can be reached at briand@iese.fhg.de.

WALCÉLIO L. MELO is Lead Researcher in the Software Engineering Group in the Centre de Rescherche Informatique de Montréal in Canada. He can be reached at wmelo@crim.ca.

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© ACM 0002-0782/96/1000 $3.50