Technical Report, Univ. of Maryland, Dep. of Computer Science, College Park, MD, USA 20742. Jan. 1995.

Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented Systems

Walcélio L. Melo
University of Maryland
Institute of Advanced Computer Studies
College Park, MD 20742 USA
melo@umiacs.umd.edu

Lionel C. Briand
CRIM
1801 McGill College Av.
Montréal (Quebec),
H3A 2N4, Canada
Lionel.Briand@crim.ca

Victor R. Basili
University of Maryland
Institute of Advanced Computer Studies and Computer Science Department
College Park, MD 20742 USA
basili@cs.umd.edu

July 6, 1995

Paper Accepted for Publication in the *Communications of the ACM.*

Abstract

This paper 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 OO systems. Reuse is assumed to be a very effective strategy for software industry to build high-quality software. However, there is currently very little empirical information about what we can expect from reuse in terms of productivity and quality gains. This also applies to OO development which is supposed to facilitate reuse. Our experiment is one step towards a better understanding of the benefits of reuse in an OO framework, considering currently available technology. Data was collected, for four months, on the development of eight medium-size management information systems with equivalent requirements. All eight projects were developed using the Waterfall Software Engineering Life Cycle Model, an Object-Oriented (OO) design method and the C++ programming language. This study indicates significant benefits from reuse in terms of reduced defect density and rework as well as increased productivity.

Key-words.

Reuse; Metrics; Object-Oriented Software Development; Software Quality and Productivity; Rework;

1. Introduction

Software reuse can help address the software crisis by helping produce quality software more quickly. Software reuse is the process of using existing software artifacts instead of building them from scratch [Krueger, 1992]. Broadly speaking, the reuse process involves the following steps: (1) selection of reusable artifact, (2) its adaptation to the purpose of the application, and finally (3) its integration 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. Much research, e.g. [Agresti&McGarry, 1987; Brooks, 1987; Thomas *et. al.*, 1992], has suggested that software quality can be improved by reusing quality software artifacts. Some work has also hypothesized that software reuse is an important factor in the reduction of maintenance costs, since when reusing quality objects, the time and effort required to maintain software products can be reduced [Basili, 1990; Rombach, 1991]. This is why the reuse of software products, software processes and other software artifacts is considered the technological key to enabling the software industry to attain the required levels of productivity and quality [Basili&Rombach, 1991].

The goal of this paper is to assess the impact of product reuse on software quality and productivity in the context of Object-Oriented (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 can actually be achieved with currently available technology. Therefore, there is a need for a better understanding of the potential benefits of OO for reuse as well as its current limitations. In this paper, the quality attributes we consider are: rework effort and number/density of defects found during the testing phases. The results that we present are based on data collected on eight medium-size projects.

This paper is organized as follows. Section 2 presents an overview of the projects analyzed as well the methodology we used to run this study. Section 3 presents the data collected together with the statistical analysis of the data. Finally, section 4 concludes the paper by presenting lessons learned and future work.

2. Description of the Study

We first present the framework of our study and then its goals and assumptions are outlined. Finally, the metrics we need are identified and defined.

2.1 Framework

The study was run for four months (from September to December, 1994). The population under study was a graduate level class offered by the Department of Computer Science at the University of Maryland. The students were not required to have previous experience or training in the application domain or OO methods. All students had some experience with C or C++ programming and relational databases.

The students were randomly grouped into teams. Each team developed a medium-size management information system that supports the rental/return process of a hypothetical video rental business and maintains customer and video databases.

The development process was performed according to a sequential software engineering life-cycle model derived from the Waterfall model. This model includes the following phases: Analysis, Design, Implementation, Testing, and Repair. At the end of each phase, a document was delivered: Requirement specification, design document, code, error report, and finally, modified code, respectively. Requirement specification and design documents were checked in order to verify if they matched the system requirements. Errors found in these two first phases were reported to the students. This guaranteed that the implementation began with a correct OO analysis/design. The testing phase was accomplished by an independent group composed of experienced software professionals. This group tested all systems according to similar test plans and using functional testing techniques. During the repair phase, the students were asked to correct their system based on the errors found by the independent test group.

OMT, an OO Analysis/Design method, was used during the analysis and design phases [Rumbaugh *et. al.*, 1991]. The C++ programming language, the GNU software development environment, and OSF/MOTIF were used during the implementation. Sparc Sun stations were used as the implementation platform. Therefore, the development environment and technology we used are representative of what is currently used in industry and academia.

Building up general and domain specific libraries

One of the most important requirement for this study was to provide a library of reusable components. (Otherwise, how could we establish the impact of software reuse on software quality and productivity?) Two kinds of libraries could be provided: Generic or domain specific. We provided both. Since most software organizations are specialized in specific application domains, most of them should have domain specific reusable components available for new development. In addition, generic components handling common data structures and providing high-level access to databases are usually available, at least within an informal setting.

The following libraries were provided to the students:

a) *MotifApp*. This public domain library provides a set of C++ classes on top of OSF/MOTIF for manipulation of windows, dialogs, menus, etc. [Young, 1992]. The MotifApp library provides a way to use the OSF/Motif library in an OO programming/design style.

b) *GNU library*. This library is a public domain library provided in the GNU C++ programming environment. Its contains functions for manipulation of string, files, lists, etc.

c) *C++ database library*. This library provides the implementation in C++ of multi-indexed B-Trees.

No special training was provided for the students in order to teach them how to use these libraries. However, a tutorial describing how to implement OSF/Motif applications was given to the students. In addition, a C++ programmer, familiar with OSF/Motif applications, was available to answer questions about the use of OSF/Motif and MotifApp libraries. A hundred small programs exemplifying how to use OSF/Motif widgets were also provided. Finally, the code sources and the complete documentation of the libraries were made available. It is important to note that the students were not mandated to use the libraries and, depending on the particular design they adopted, different reuse choices were expected.

Assuming that a conventional organization would have past experience on developing business application, we also provided a specific domain application library in order to make our experiment more representative of the "real world". This library implemented the graphical user interface for insertion/removal of customers and was implemented in such a way that the main resources of the OSF/Motif and MotifApp libraries were used. Therefore, this library contained a small part of the implementation required for the development of the rental system.

2.2. Experiment Goals and Assumptions

In order to define the metrics to be collected during the experiment we used the Goal/Question/Metric (GQM) paradigm [Basili&Weiss, 1984]. This study has the following goal:

Analyze:	Reuse in an OO Software Development Process
for the purpose of:	evaluation
with respect to:	rework effort, defect density and productivity
from the viewpoint of:	organization.

In other words, our objective was to assess the following assumptions in the context of OO systems developed under currently available technology:

1) A high reuse rate results in a lower likelihood of defects.
2) A high reuse rate results in lower rework effort (i.e., lower effort to repair software products.)
3) A high reuse rate results in higher productivity.

According to the QGM paradigm, we should define at this stage a set of questions pertinent to the defined experimental goal and a set of metrics allowing us to devise answers to those questions. Due to a limitation of space, we will not present the complete GQM and will only present in this paper the metrics we have derived. However, it is important to note that the metrics described in the next section were derived by following the GQM methodology [Basili&Rombach, 1988].

2.3. Metrics

2.3.1 Size

The size of a system S is a function *Size(S)* that is characterized by the properties: *Size* cannot be negative (property Size.1), and we expect it to be null when a system does not contain any component (property Size.2). When components do not have elements in common, we expect *Size* to be additive (property Size.3).

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

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

where Size(c) is equal to the number of lines of code of the component c [Fenton, 1991].

These concepts are fully formalized in [Briand *et. al.*, 1994].

2.3.2. Reusability

The “amount” of reused code in a system S is a function *Reuse(S)* that is also characterized by the properties Size.1 - Size.3, that is *Reuse(S)* is a type of 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, like classes and inheritance. For instance, consider a class C which is included in a system S. There are several cases:

1) C belongs to the library:

   In this case we have verbatim reuse, i.e., 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 which allows inheritance, all ancestors of C have also to be included into S. As all C ancestors also belong to the library, including a library class may trigger a large "amount" of verbatim reuse.

2) Class C is a new class which has been created by specializing, through inheritance, a library class, LC. This is in fact a variation of first case, i.e., the class LC and all its ancestors will be included into S and will be dealt with similarly to verbatim reuse.

3) Class C has been created by changing an existing class, EC. Reuse could be logically estimated as:

   Reuse(C) = %Change * Size(C)

   where %Change represents the percentage of C deleted or modified.

   However, this kind of data 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 component was labeled: "Extensively modified" and in the latter case: "Slightly modified".

   When working at the project level, reuse rates were computed based on the following approximations:

   a) Extensively modified: Reuse(C) = 0

   b) Slightly modified: Reuse(C) = Size(C)

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

4) C has been created from scratch. In this case, the amount of reuse of the class C is zero:

   Reuse(C)=0.

   Now, let us assume an operator called *Classes* which when applied to a system S, gives all classes of the system S, such that:

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

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

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

   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 following property: ReuseRate(S) † 1,

2.3.3. Effort

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

- Person-hours across development activities. This includes:

  ° Analysis. The number of hours spent understanding the concepts embedded in the system before any actual design work. This activity includes requirements definition and requirements analysis. It also includes the analysis of any 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, defining object interactions, etc. The time spent reviewing design material, such as 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

- *Person-hours across errors (rework).* This includes the number of hours spent on (1) isolating an error and (2) correcting it.

2.3.4. Productivity

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

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

where, in this study:

- Size(S) is instantiated as the number of lines of code delivered in the system S. Other size measures (e.g., function points) could have been used but this one fulfilled our requirements and could be collected easily. More importantly, we are looking at the relative size of systems addressing similar requirements and, therefore, of similar functionality.

- DE(S) (development effort) is defined as the total number of hours that a group spent on analyzing, designing, implementing and repairing the system S.

2.3.5. Number of Defects

Here we analyze the number and density of defects found for each system/component. We will use the term defect as a generic term, to refer to either an error or a fault. Errors and faults are two pertinents ways to count defects and we believe they should be both considered in this 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 within 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 across the test phases.

2.4. Collecting data

In our case we used the approach proposed in [Basili&Weiss, 1984] 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. These forms have been tailored from those used by the Software Engineering Laboratory [Heller et. al, 1992].

- Personnel Resource Form.
- Component Origination Form.
- Error Report Form.

In the following sections, we comment on the purpose of those three forms and the data collection processes.

2.4.1 Personnel Resource Form

This form is used to gather information about the amount of time the software engineers spent on each software development phase. Section 2.3.3 provides the taxonomy of phases that were used.

2.4.2 Component Origination Form

This form is used to record information that characterizes each component in the project under development at the time it gets into configuration management. This form is used to capture whether the component has been developed from scratch or has been developed from a reused component. In the latter case, we collected the amount of modification (none, small or large) that was needed to meet the system requirements and design as well as the name of the reused component. By small/large, we mean that less/more than 25% of the original code has been modified, respectively. This is a simplification that has been discussed above.

2.4.3 Error Report Form

This form was used to gather data about (1) the errors found during the testing phase, (2) components changed to correct such errors, and (3) the effort in correcting it. This latter item includes:

- how long it took to determine precisely what change was needed, the effort required for understanding the change or finding the cause of the error, locating where the change was to be made, and determining that all effects of the change were accounted for.

- how much time was needed to implement the correction: design changes, code modifications, and regression testing.

3. Results

The following section describes the results of the study with respect to code reuse. As mentioned earlier, this analysis focuses on three points: defect density, rework, and productivity. The next three subsections cover these issues. In the first two subsections, we present the results in two ways: (1) assessing the differences in quality and productivity across reuse categories as they are defined in Section 2.3.2 (new, extensively modified, slightly modified, verbatim), and (2) compute an approximate project reuse rate and assess its association with project quality and productivity. In the first form of analysis, projects are considered as separate and different entities while in the second form of analysis, trends across projects are analyzed and the analysis therefore assumes the projects to be comparable. In addition, the first type of analysis will help us justify the definition of the reuse metric we used for this study (Section 2.3.2).

3.1 Reuse versus Defect Density

The first analysis compares reused and newly created code from the perspective of defect density. For reasons provided below, we look at defects according to two definitions: errors and faults. In our study, an *error* is assumed to be represented by a single error report form; a *fault* is represented by a physical change to a component. We used a simple measure of component size: lines of code, but other size measures could have been used for the same purpose. Because of the inherent fuzziness of such concepts, we realize that different results could have been obtained according to different scales of measurement on which these concepts could have also been defined. However, since we are 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.

3.1.1 Relationship between Defect density and component origin

We first examine the relationship between components and defect density to see if reused components have a lower defect rate. In addition, we use this analysis as an opportunity to evaluate the meaningfulness of our ordinal reuse measure and assess its impact on defect density.

Component Origin	**No. of Comp.**	**No. of LOC**	**No. Faults**	**Fault Density**	**No. of Errors**	**Error Density**	**Rework**
New	177	25642	247	9.63	157	6.11	336.35
Extensively Modified	79	15165	93	6.13	74	4.89	160.04
Slightly Modified	45	6685	11	1.57	10	1.50	22.5
Reused Verbatim	135	25388	6	0.24	2	0.06	3
All Components	436	72880	356	4.88	243	3.33	521.89

Table 1: Errors / Fault densities and rework in each component origin category for all projects.

Table 1 shows the error and fault densities (errors and fault per thousand of lines of code) observed in each of the four categories of component origin. Apparently, fewer defects were found in reused code. For example, error density (#Errors/KLOC) was found to be only 0.06 in the code reused verbatim, 1.50 in the slightly modified code, 4.89 in the extensively modified, and 6.11 in the newly developed code.

However, these differences should be assessed statistically, i.e., the level of significance of these trends should be calculated. In order to perform this statistical analysis, defect densities were computed for each project and each reuse category, i.e., defect density was computed for each project's sub-set of components 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 is therefore matching one observation in each of the other reuse categories since they correspond to the same system, satisfy a similar set of requirements, have been developed by 8 teams from the same population, and have similar complexities and comparable functionalities (as demonstrated later). Conceptually, it is *almost* equivalent to looking at the characteristics of identical sets of components developed with different reuse rates.

Therefore, we used a non-parametric test that compares pairs of observations across reuse categories (Wilcoxon matched-pairs signed rank test or Wilcoxon T test [Devore, 91]). The logic 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 will be that there is no difference between pairs of scores for the population from which the sample of pairs is drawn. If this is true, we would expect to have similar numbers of negative and positive differences and similar magnitudes of differences. Therefore, the Wilcoxon T test takes into account both the direction and magnitude of differences between scores (i.e., defect densities) to determine if the null hypothesis is reasonable. Thus, by using this test, we can obtain a statistical comparison of any project characteristic (i.e., defect density, etc.) across component reuse categories. Such a test does not assume all projects are comparable and is robust to outliers, i.e., extreme differences of scores between pairs. This last property is particularly important when considering small samples.

We first look 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 component 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 the diamond width is proportional to the category sample size.) Counting faults is conceptually equivalent to weighting errors according to their dispersion across components since we count as many faults as there are components affected by a given error. Recall that the question here is: does reuse reduce fault-proneness? In other words, the null hypothesis is: reused and non-reused components are, on the average, of “similar fault-proneness”. However, in order to interpret the results right, we have to check that the internal structure of the components, 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, function calls, etc.) and were able to determine that the distributions across reuse categories were not statistically different. (These measures were extracted using the Amadeus tool [Amadeus, 1994]).

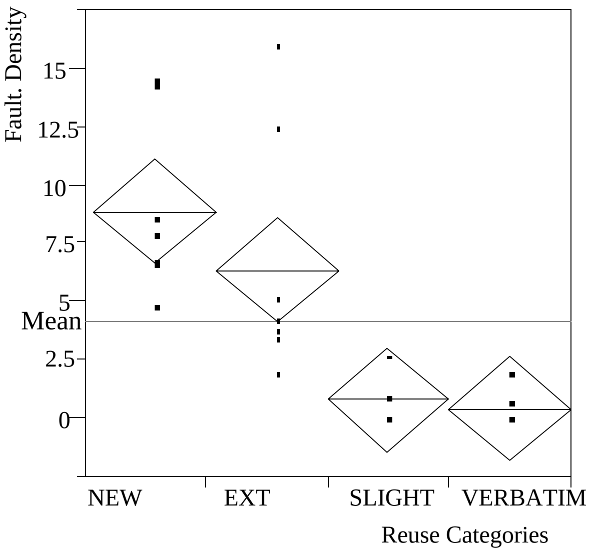

Figure 1: Distribution and mean of project fault densities across reuse categories

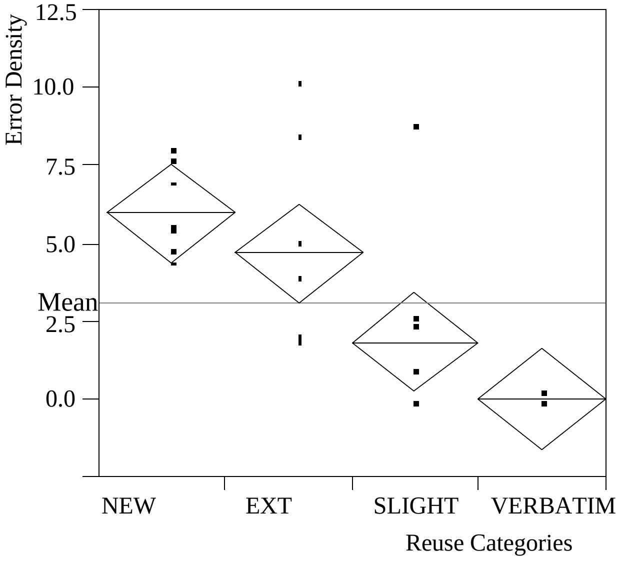

Figure 2: Distribution and mean of project error density per reuse category

Table 2 shows the paired statistical comparisons of fault densities between reuse categories. We assumed significance at the 0.05 level, i.e., if the p-value is greater than 0.05, then 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 not support for the fact that there is an observable difference between verbatim reuse and slightly modified, and between extensively modified and new. 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.

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

Table 2: Levels of significance - fault density per reuse category.

We used the same approach in order to obtain a statistical comparison of component *error density* per component category. Error density is more complicated to compute with respect to reuse categories as an error may trigger changes in several components from different categories. Thus we count the number of errors per component by using the following equation:

$$\#\text{Errors in } C_i = \sum_{\text{all errors affecting } C_i} 1/\#\text{components\_affected}$$

and then, for each component category, the sum of the components' number of errors is computed for each project and divided by the sum of the sizes of those components.

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 components affected belonged to a same reuse category.

In this case, all errors are considered with equal weight. The distributions are shown in Figure 2 and the Wilcoxon T test results are shown in Table 3.

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

Table 3: Levels of significance - error density per reuse category

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

3.1.2 Relationship between Project Reuse Rate and Error Density

Similar to what was done in the previous section, we would like to verify the hypothesis that, the higher the project reuse rate, the lower the number of errors. For the sake of simplification, only verbatim reused and slightly modified components are considered as "reused components" in order to compute the reuse rate per project. This approximation is to some extent justified by the results of the previous section that show extremely different trends for slightly and extensively modified components. We see this analysis as complementary to the last section’s analysis since the determination of the relationship between reuse rate and error density allows us to quantify the impact of reuse. The drawbacks of this new analysis, however, are that all projects are assumed to be comparable and that the results can be easily biased by outliers. Table 4 provides a complete overview of the projects' data: Number of lines of code delivered at the end of the implementation phase, rate of reuse per project, the project error density and, finally, rework.

Project	No. of LOC	Reuse Rate	Error Density	Rework
1	23354	71.83	1.03	51
2	5068	2.23	6.51	71
3	9735	31.44	4.31	92
4	8543	18.08	3.86	72
5	8173	40.05	3.18	59
6	6368	48.67	3.93	51
7	6571	64.01	2.28	31
8	5068	0.00	8.68	93

Table 4: Overview of Projects' Data.

The average rate of reuse is approximately 34%, with a maximum of 72%. There appears to be a strong linear relationship between reuse rate and project error density that is significant at the 0.0018 level (F-test). The estimated intercept and slope are 6.55 and -0.095, respectively. That means that, when there is no reuse, error density should be expected to be around 6.55 and each additional 10 percentage points in the reuse rate decreases this density by nearly 1. No outlier seems to be the cause of a spurious correlation and, therefore, this result should be meaningful (Figure 3).

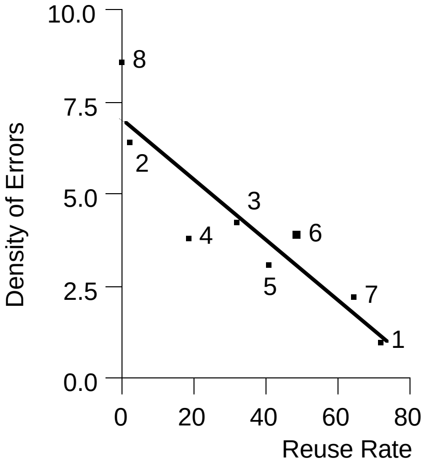

Figure 3: Linear relationship between error density and reuse rate

Thus, the results presented above support the assumption that reuse in OO software development results in a lower likelihood of defects.

3.2 Reuse versus rework

[Jones, 1986] identifies rework as a major cost factor in software development. According to Jones, rework on the average accounts for over 50% of the effort for large projects. Reuse of previously developed, reviewed, and tested components could result in easy-to-maintain components and consequently, reuse should decrease the rework effort. In this section, we first compare rework effort on reused and newly created components. Then we check whether the total amount of reuse per project is related to a reduction in the rework effort.

3.2.1 Component Rework versus Reuse Categories

Here, we are interested in answering the following question: Is the effort spent to repair reused components lower than that to repair components created from scratch or extensively modified?

We looked at three different metrics to answer this question: (1) total amount of rework in each component reuse category, (2) rework normalized by the size of the components belonging to each reuse category, and (3) rework normalized by the number of faults that were detected in the component of each reuse category. Distributions and means are shown in Figures 4, 5 and 6, respectively.

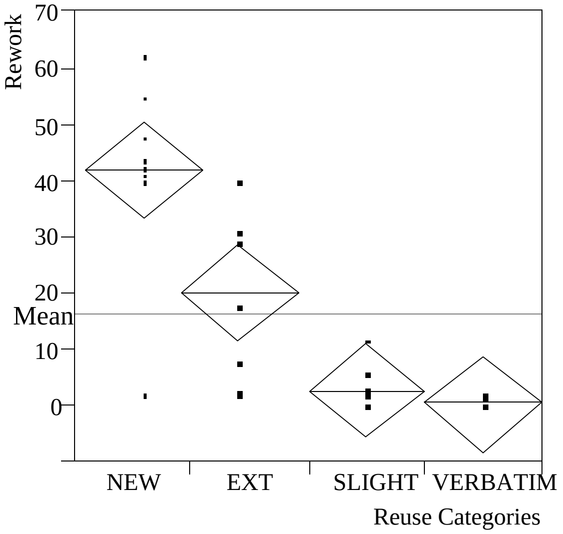

Figure 4: Distribution and mean of project rework per reuse category

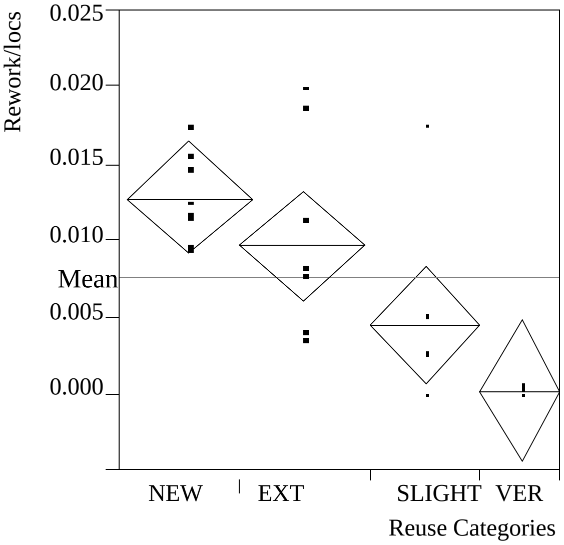

Figure 5: Distribution and mean of project rework density per reuse category

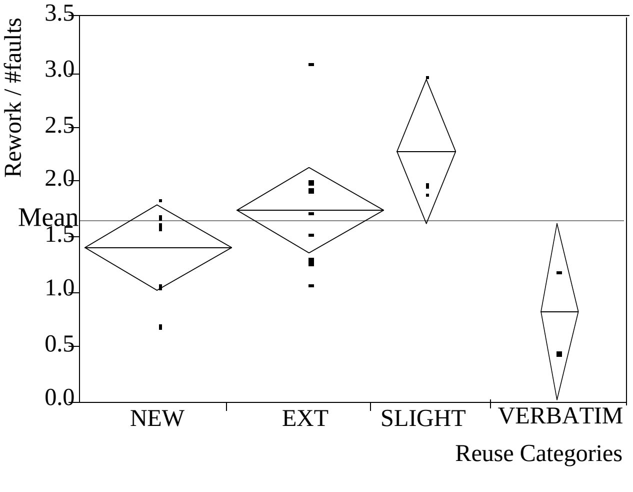

Figure 6: Distribution and mean of project rework difficulty per reuse category

These different metrics allow us to look at rework from different perspectives:

(1) Captures the total cost of rework and is therefore expected to be somewhat associated with the size of the components and the number of faults in each reuse category.

(2) Allows us to look at rework without considering the relative amount of code in each reuse category. Thus, we get an insight into the relative cost of debugging and perfecting code in each reuse category.

(3) Allows us to look at the expected difficulty to repair a fault in the various reuse categories.

Before any thorough statistical analysis, a look at the distributions seems to indicate that (1) and (2) are different across reuse categories**.**

Based on these results, we can run again, as for defect density analysis, a Wilcoxon signed rank test [Devore, 1991] on the data collected in order to answer the question formulated at the beginning of this section. In other words, instead of using defect density per reuse category as scores, we use total amount of rework per reuse category. The results for metric (1) are shown in Table 6.

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

Table 6: Rework per Reuse Category

Based on Table 6, we can conclude that reuse reduces the amount of rework even when the code is extensively modified. Again, there is no observable difference between verbatim reused and slightly modified components. These results show that, from the perspective of total rework, extensively modified code might still bring benefits and that, once again, slightly modified code is nearly as good as code reused verbatim.

Unfortunately, because some projects have no verbatim or slightly reused components, the number of data points in these categories becomes too small for applying a Wilcoxon T test to metrics (2) and (3). In those cases, we can only look at the difference between new and extensively modified components. No significant difference can be observed for both metrics. In order to facilitate statistical testing and, at least, get a rough answer for (2), we merge the new and extensively modified categories and the slightly and verbatim reused categories, respectively. With respect to (2), a statistically significant difference can be observed at a 0.03 level. Therefore, reused lines of code appear to be less expensive to debug, i.e., a lower "rework density".

As a last attempt to look at change difficulty (metric (3)), we performed an analysis at the component level where rework effort per component was normalized by the number of faults detected and corrected in these components. No significant differences in distribution could be observed across reuse categories.

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

3.2.2 Relationship between Project Reuse Rate and rework

To complement previous section results, we would like to verify whether the larger the project reuse rate, the lower the 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 at a 0.02 level (F-test). The estimated intercept and slope are 86.46 and -0.62, respectively. That means that, where there is no reuse, rework effort should be expected to be around 86.46 man-hours and for each additional 10 percentage points in reuse rate, rework effort will decrease by nearly 6.3 man-hours. These results are of course specific to the system requirements implemented in this study and could be generalized as follows: each additional 10 percentage points in reuse rate decreases rework by 7%. Again, no outlier seems to be the cause of a spurious correlation (see Figure 7).

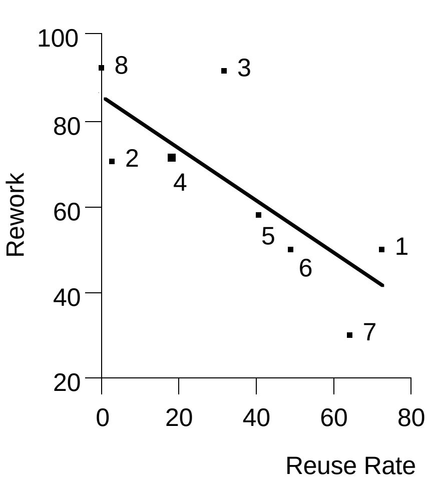

Figure 7: Linear relationship between rework and reuse rate

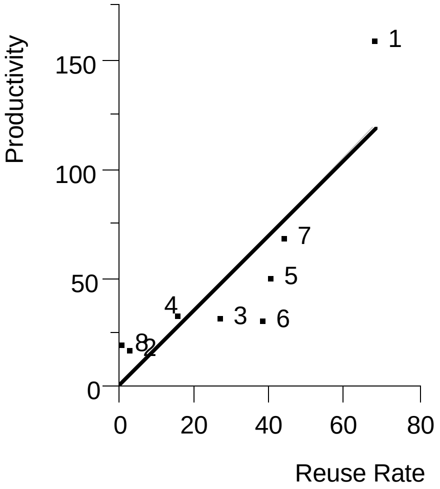

Figure 8: Linear relationship between productivity and reuse rate

In order to better capture the concept of rework, we believe 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 accurately this information with the data collection processes 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 compare rework normalized by the number of faults at the component level.

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

3.3. Project Reuse Rate versus Productivity

Reuse has been advocated as a means for reducing development cost. For example, in [Boehm&Papaccio, 1988], reuse of components is identified as one of the most attractive strategies for improving productivity. As productivity is often considered to be an exponential function of software size, a reduction in the amount of software to be created could provide a dramatic savings in development costs [Boehm, 1984]. The question is now, does reuse improve productivity despite change and integration costs?

Table 7 shows for each project analyzed: (1) the number of lines of code delivered at the end of the life-cycle, (2) the number of lines of code reused (verbatim plus slightly modified components), (3) the productivity, and (4) the reuse rate. Note that the reuse rate and LOC delivered column are different from the data showed in Table 4. This difference stems from the fact that Table 7 presents the results at the end of the life-cycle, i.e., after the errors have been fixed, whereas Table 4 presents the data collected at the end of the implementation phase.

Based on data presented in Table 7, we can conclude that there is also a strong linear relationship, which is statistically significant at a 0.007 level (F-test), between productivity and reuse rate (see figure 8). The estimated intercept and slope are 1.68 and 1.73, respectively. That means that, when there is no reuse, productivity should be expected to be around 1.68 and each additional 10 percentage points in the reuse rate increases productivity by nearly 20 LOC per hour. However, in this case, one data point appears to be an outlier and may explain such a good level of significance. Since only one project shows a really high reuse rate, this trend may appear more significant than it is and these results should be considered rough approximations.

Project	LOC Delivered	Reused	Productivity	Reuse Rate
1	24698	16776	159.34	67.92
2	5105	113	18.23	2.21
3	11687	3061	32.01	26.19
4	10390	1545	34.3	14.87
5	8173	3273	51.4	40.04
6	8216	3099	31.12	37.71
7	9736	4206	69.54	43.2
8	5255	0	19.9	0

Table 7: Data collected after the errors have been fixed.

In order to explain some of the variations observed on this scatterplot, we performed some qualitative analysis of the process that had been followed, the teams involved, and the design strategies adopted in each project. Project 6 appears to have a particularly low productivity, considering its reuse rate. This was explained by the particularly sophisticated graphical user interface that this group designed. In the context of the requirements we provided to the students, this could be considered "gold-plating". The team in Project 3 had no previous experience with respect to GUI's and learning the basics was perceived as a significant effort. In project 1, by heavily reusing the available C++ database library, the team reached a very high productivity.

4. Conclusion

This paper has provided significant results showing the strong impact of reuse on product productivity and, more particularly, on product quality in the context of object-oriented management information 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 estimations by managers and can be used as a baseline of comparison in future reuse studies for the purpose of evaluating reuse processes and technologies.

This study can be replicate in the industry or academia. In industry, replicating this study can, for example, help managers decide whether it is worth investing in particular object-oriented technologies in order to improve software quality and/or productivity. In academia, replicating this study can help test different OO methods or compare the advantages of such methods against traditional development methods.

Future work includes the 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 collection and, therefore, the design of tools monitoring the changes to code and design documents. Finally, it would be interesting to refine our comparison of the internal component characteristics across reuse categories by using more specific OO metrics [Abreu&Carapuça, 1994]. We need to better characterize the impact of reuse on the system size, complexity, coupling, and cohesion [Fenton, 1991].

Acknowledgements

We want to thank (1) Dr. Gianluigi Caldiera for teaching the OMT method; (2) Jyrki Kontio, Carolyn Seaman and Barbara Swain for their suggestions that helped improve both the content and the form of this paper; (3) and the students of University of Maryland for their participation on this study.

References

F. B. Abreu and R. Carapuça (1994). "Candidate metrics for object-oriented software within a taxonomy framework". In *Journal of System and Software*, 26(1):87–96.

Amadeus Software Research, Inc. (1994). *Getting Started with Amadeus*. Amadeus Measurement System.

W. W. Agresti and F. McGarry (1987). The Minnowbrook workshop on software reuse: A summary report. In W. Tracz, editor, *Software Reuse: Emerging Technology*. IEEE Press.

V. Basili (1990). Viewing maintenance as reuse-oriented software development. In *IEEE Software*, 7(1):19–25, Jan.

V. Basili and H. D. Rombach (1988). "The TAME project: Towards improvement-oriented software environments". In *IEEE Trans. on Software Engineering*. 14(6):758–773.

V. Basili and H. D. Rombach (1991). "Support for comprehensive reuse". In *IEE Software Engineering Journal*. Sept. pp. 303-316.

V. Basili and D. M. Weiss (1984). "A methodology for collecting valid software engineering data". *IEEE Trans. on Software Engineering*, 10(6). November.

B. W. Boehm (1984). Software engineering economics. *IEEE Trans. on Software Engineering.* SE-10(1), January.

B. W. Boehm and P. N. Papaccio (1988). Understanding and controlling software costs. *IEEE Trans. on Software Engineering*, 14(10), October.

L. Briand; S. Morasca; V. Basili (1994). "*Goal-Driven Definition on Product Metrics Based on Properties*". CS-TR-3346, University of Maryland, Dep. of CS, College Park, MD, 20742.

F. P. Brooks (1987). No silver bullet: essence and accidents of software engineering. In *Computer*, 20(4), April.

J. Devore (1991). *Probability and Statistics for Engineering and Sciences*. Brooks/Cole Publishing Company

N. E. Fenton (1991). *Software Metrics: A Rigorous Approach*. Chapman&Hall.

T. C. Jones (1986). *Programming Productivity*. McGraw-Hill.

C. W. Krueger (1992). "Software reuse". In *ACM Computing Surveys.* 24(2):131-183.

G. Heller; J. Valett; M. Wild (1992). *Data Collection Procedure for the Software Engineering Laboratory (SEL) Database.* SEL Series, SEL-92-002.

H. D. Rombach (1991). Software reuse: a key to the maintenance problem. In *Information and Software Technology Journal*, 33(1), Jan/Feb.

J. Rumbaugh; M. Blaha; W. Premerlani; F. Eddy; W. Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice-Hall.

W. Thomas, A. Delis, V. Basili (1992). An evaluation of Ada source code reuse. In *Proc. of the Ada-Europe Int'l Conf.*, Zandvoort, The Netherlands, June.

D. A. Young (1992). *Object-Oriented Programming with C++ and OSF/MOTIF*. Prentice-Hall.