



NORTH-HOLLAND

An Analysis of Errors in a Reuse-Oriented Development Environment

William M. Thomas

Department of Computer Science, University of Maryland, College Park, Maryland

Alex Delis

Department of Computer and Information Science, Polytechnic University, Brooklyn, New York

Victor R. Basili

Department of Computer Science, University of Maryland, College Park, Maryland

Component reuse is widely considered vital for obtaining significant improvement in development productivity. However, as an organization adopts a reuse-oriented development process, the nature of the problems in development is likely to change. In this article, we use a measurement-based approach to better understand and evaluate an evolving reuse process. More specifically, we study the effects of reuse across seven projects in narrow domain from a single development organization. An analysis of the errors that occur in new and reused components across all phases of system development provides insight into the factors influencing the reuse process. We found significant differences between errors associated with new and various types of reused components in terms of the types of errors committed. In addition, we identified differences when errors are introduced and the effect that the errors have on the development process. © 1997 Elsevier Science Inc.

1. INTRODUCTION

Reuse has been advocated as a technique with great potential to increase software development productivity, reduce development cycle time, and improve product quality (Agresti and McGarry, 1987; Brooks, 1987; Boehm and Papaccio, 1988). However, reuse

will not just happen; rather, components must be designed for reuse, and organizational elements must be in place to enable projects to take advantage of the reusable artifacts.

Basili and Rombach (1991) present a framework of comprehensive support for reuse, including organizational and methodological properties necessary to maximize the benefit of reuse. For reuse to attain a significant role in an environment, organizational changes must be made to facilitate the change in development style. Maintaining a library of reusable parts may require resources, including personnel, hardware, and software. While increasing the amount of reuse in an environment may reduce certain development activities (e.g., code creation), it will also require additional effort in other activities (e.g., searching for components). With respect to product quality, it is also clear that "reused" does not imply "defect-free". An investigation into the benefits of reuse in the NASA Goddard Space Flight Center (NASA/GSFC) showed that even among components that were intended to be reused verbatim, while their error rate was an order of magnitude lower than newly created code, the error rate is still significant (Thomas et al., 1992). By analyzing the nature of the defects in the reuse process, one can tailor the process appropriately to best achieve the organization's goals.

There have been several studies into techniques to stock an initial reuse library (Caldiera and Basili, 1991; Dunn and Knight, 1993). One factor to be

Address correspondence to Prof. Victor R. Basili, Department of Computer Science, University of Maryland, College Park, MD 20742.

considered is the structure of the candidate reusable component. Selby (1988) investigated various characteristics of new versus reused code in a large collection of FORTRAN projects. Basili and Perricone analyzed tradeoffs between creating a component from scratch versus modifying an existing component (Basili and Perricone, 1984). In this article, we extend these studies by investigating the nature of errors occurring in a reuse-oriented development environment, and drawing conclusions about their impact in such an environment. In particular, we analyzed a collection of eight medium-scale Ada projects developed over a five-year period in the NASA/GSFC with respect to the defects found in newly developed and reused components. The goal of the study was to learn about the nature of problems associated with reuse-oriented software development, thereby allowing for improvement of the reuse process. We found significant differences between errors associated with new and with various types of reused components in terms of when errors are being introduced, the effect that they have on the development process, and the type of error being committed. We also found a number of similarities and differences with the findings of other investigations into component reuse.

This article is organized as follows. Section 2 provides a brief overview of reuse-oriented software development, while Section 3 gives background about using error analysis for process improvement. Section 4 describes the goals of the study and the data analyzed. The findings from our analysis are presented in Section 5, and Section 6 summarizes and identifies the major conclusions.

2. REUSE-ORIENTED SOFTWARE DEVELOPMENT

Reuse has been cited as a technology with the potential to provide a significant increase in software development productivity and quality. For example, Jones (1984) estimates that only 15% of the developed software is unique to the applications for which it was developed. Reduced development cost is not the only benefit of reuse—in fact, the greatest benefit from reuse may be its impact on maintenance (Lanergan and Grassol, 1984; Rombach, 1991). The potential for substantial savings from reuse clearly exists. Unfortunately, achieving high levels of reuse still remains an elusive task. A number of issues must be addressed to effectively increase the level of reuse in an organization, including the forms of reuse, and language and organizational support to encourage reuse.

2.1 Types of Reuse

In this study, we examined three modes of reuse:

- verbatim reuse, in which the component is unchanged;
- reuse with slight modification, in which the original component is slightly tailored for the new application;
- reuse with extensive modification, in which the original component is extensively altered for the new application.

While there is a clear distinction between verbatim reuse and reuse via modification, distinguishing between slight modification and extensive modification is a more subjective matter. Our intent is to distinguish between cases where a component is left essentially intact, but needs a minor change for a new application, and cases where a component is significantly altered for a new use. The three types of reuse and their expected impact on development are described in the following paragraphs.

Intuitively, verbatim reuse appears to hold the greatest benefit to software development. Development effort is minimized and verification effort is reduced because the component has previously been developed, tested, and used. There may be an increased cost in integration effort, as the reused component may not squarely fit in the new system, and the developers may not be as familiar with the reused component as they would be with a custom component.

Another means of reuse is achieved by slight modification of an existing component. Here a component remains for the most part unchanged but is adapted slightly for the new application. For example, a sort routine may be modified to sort a different type of objects. An improvement in terms of reduced development effort and increased quality is expected, although perhaps not of the same degree as in the reused verbatim components. Again, the integration of modified components may be more difficult than that of newly created components; however, because the modified components may be adapted to better match the application, the integration may not be as difficult as with the verbatim reused components. As with verbatim reuse, there may be new errors introduced in the component selection process. However, since the developer does have a greater understanding of the implementation of the modified component, one is more likely to detect that error earlier than if the component was reused verbatim.

Our third category of reuse occurs through extensive modification of an existing component. For example, one may want to change the underlying representation of a particular type while maintaining the operations on the type. If the component was not designed with the representation isolated in the implementation, this may require changes throughout the component. Reuse in this manner is likely to be beneficial only if the component is of a sufficient size and complexity to justify modification as opposed to simply creating a new component from scratch. Since much of the component is new, in many ways, this type of reuse may appear similar to new development. However, there are some important distinctions. The number of coded lines is likely to be reduced relative to newly developed code, so one might expect a decrease in error density. On the other hand, the extensive modification activity may be more error prone than standard component creation, since the original abstraction is being significantly altered. This mode of component creation may result in more of a "hack" than a well-conceived component. New types of errors may arise, such as removing too much or not enough of the old component.

2.2 Language Issues in Software Reuse

The Ada programming language contains a number of constructs that encourage effective reuse, including packages and generics (Ichbiah, 1985; Wolf et al., 1985; Gargaro and Pappas, 1987; Ebel and Genillard, 1990). A package is used to group a collection of declarations, such as types, variables, procedures, and functions. The package construct allows for the encapsulation of related entities, encouraging the creation of well-defined abstractions such as encapsulated data types. For example, a stack package of a particular type can be created, containing the element type and operations such as push and pop. Through a simple modification of the element type, the package can be adapted to support operation on a different type. This would enable one to move toward the second type of reuse, tailoring the component slightly to suit the new application.

Ada's generic construct provides more support for verbatim reuse, as it enables the creation of more abstract entities. A generic program unit is a template for a module. Instantiation of the generic program unit yields a module. The generic units may be parameterized, i.e., they may require the user to supply types or operations to create a module. This provides a great deal of flexibility in their use. For example, one may parameterize the stack package

such that the user must supply the element type to create an instance of the stack. The generic stack can then be used without modification in support of a number of different types.

High levels of reuse may be achieved in languages without such features; however, the approach taken to achieve such reuse will be different. Such differences were reported in a study comparing FORTRAN and Ada reuse in the NASA/SEL (Bailey et al., 1993). The Ada approach was to develop a set of generics that can be instantiated to support a variety of application types. In contrast, the FORTRAN approach was to develop a collection of libraries specific to each application type. On projects within a very narrow domain, both approaches achieved similar high levels of reuse. However, when there was a significant change in the domain, the Ada approach achieved a sizable amount of reuse (50% verbatim reuse), while the FORTRAN approach showed less than 10% verbatim reuse (Bailey et al., 1993). Thus, it would appear that the parameterized, generic approach is better suited to development in a dynamic, evolving domain.

While improved language features may help to enable reuse, they alone have not resulted in large-scale reuse in software development. There are other important factors involved—applications must be structured to allow and encourage reuse, and software organizations must be tailored to support a reuse-oriented development paradigm.

2.3 Organizational Support for Reuse

One model that integrates reuse into a development is the "component factory" organization, which is a dual-organization structure consisting of two parts: a factory organization and a project organization. The factory organization provides software components in response to requests from the various projects being developed in the project organization (Basili et al., 1992). Figure 1 illustrates the component factory concept in support of a project organization. In this setting, the development organization makes requests to the component factory to provide components to be integrated into the desired product. If the component factory is effective, the activity of component creation can be significantly reduced, and the quality of the components that are delivered to the integration team can be increased, reducing the costs of development and of rework. The key features of the component factory are the repository of the components for future reuse, and the focus on flexibility and continuous improvement. Thus, a measurement-oriented approach must be

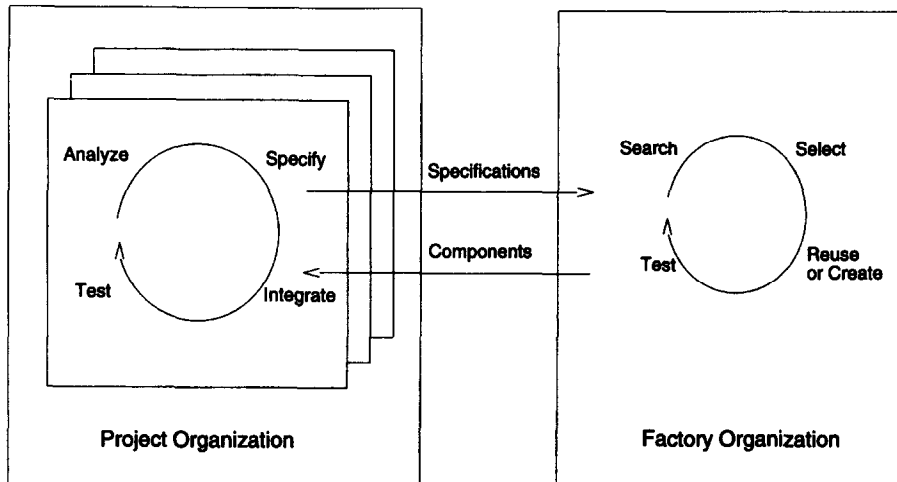


Figure 1. Interaction of a project organization with the component factory.

utilized, such as that proposed in the TAME project (Basili and Rombach, 1988), which provides an experimental view of software development, allowing for analysis and learning about the effectiveness of the new technologies.

Reuse-oriented development will require some effort to be expended in activities that are not a part of traditional software development. For example, although the component factory will allow the effort spent in component creation to be reduced, it will also require additional activity in searching for and selecting the appropriate component for the particular application. These new activities may also be a potential source of errors in the system, and thus, a source of rework effort. Introducing an activity of selecting a component from a repository may introduce new types of errors, for example, selecting a component that does not provide the intended function. The goal of error analysis is to learn about the nature of errors in the current environment so that improvement can be made (e.g., process tailoring) in subsequent projects, and feedback can be provided to the current project. In the following sections, we describe our analysis of errors in an organization that has placed a significant emphasis on reuse, with the result that it has recently achieved very high levels of reuse.

3. DESCRIPTIONS OF THE ANALYSIS

Since its origin, The NASA/GSFC SEL has collected a wealth of data from their software development (SEL, 1994). Selby performed a study on the characteristics of reused components on a collection

of FORTRAN projects from this environment (Selby, 1988), in which the level of reuse (verbatim and modified) averaged 32%. Because of the support for reuse provided by the Ada language, as discussed in Section 2.2, we chose to analyze the Ada projects in this environment. A much higher level of reuse than what was reported in Selby's study of FORTRAN projects has been achieved in these Ada projects, with total reuse (verbatim or with modification) approaching 100% and verbatim reuse reaching over 90% (Kester, 1990). These high levels of reuse have been attributed in part to the Ada language constructs and object-oriented methods (Kester, 1990; Bailey et al., 1993; Stark, 1993). More recently, however, even the FORTRAN system have been showing such high levels of reuse, although the nature of the reuse is different from reuse in the Ada development environment. The Ada approach is centered on the development of a reuse library containing generics that can be instantiated with mission-specific parameters to develop new application. In contrast, the FORTRAN approach was to develop separate libraries containing subsystems for certain mission-specific options, and these subsystems must be used in an "all or nothing" manner. As such, when new projects were developed with a major change to the application domain, the Ada approach still resulted in a sizeable amount of verbatim reuse (40%), while the FORTRAN approach saw reuse drop to less than 10% (Bailey et al., 1993).

We analyzed a collection of seven medium-scale Ada projects from a specific domain, as all are stimulators which were developed at the NASA/GSFC Flight Dynamics Division. An overview of the

Table 1. Overview of the Examined Projects

Project ID	KSTMT	Pct. Total Reuse	Pct. Verbatim Reuse	Effort (SM)
A	27.1	31	4	175
B	14.4	31	13	85
C	13.7	38	19	72
D	24.8	85	27	117
E	13.8	97	88	30
F	12.8	78	44	73
G	13.7	100	89	16

projects examined is provided in Table 1. The projects ranged in size from 61 to 184 thousand source lines, or 12.8 to 27.1 thousand Ada statements (KSTMT). They required development effort of 16 to 175 technical staff months (SM). In these projects, verbatim reuse ranged from 4–89% of the total statements, while total reuse (i.e., verbatim and modification) ranged from 31–100%.

While this environment is not organized along the lines of the Component Factory discussed in Section 2, it does have some characteristics in common with that organization. In the SEL, generalized architectures were developed explicitly to facilitate large scale reuse from project to project (Stark, 1993), so it is clear that significant effort has been applied towards the goal of reuse in the organization. As such, new systems have been developed in accordance with the packaged experience of reusable architectures, designs, and code. One aspect of the Component Factory organizations is the separate organization that produces or releases all reusable software products (Basili et al., 1992). While this feature is not present in the SEL, it is apparent that less effort is being spent on project-specific development activities. The percentage of effort spent in the Coding/Unit Test phase has dropped from 44% on an early simulator, to only 18% on one of the more recent simulators (Stark, 1993). This suggests that there is a significant leveraging of the stored experience, and as such, the observed effort on the SEL projects is becoming more in line with the profile one would expect in the Component Factory's project organization, i.e., dominated by design and testing activities.

3.1 Questions Addressed in the Analysis

Our goal is to develop an understanding of the differences between traditional development methods and reuse-oriented methods in terms of the characteristics of their errors. Increased knowledge about the types of errors in an environment can be

used to optimize the process for that environment. We developed a set of questions with which to compare newly created, modified, and reused verbatim components:

1. What is the impact of reuse on error density?

Reuse has been noted as offering the potential for substantial gains in product quality; however, the different forms of reuse investigated in this study are likely to have differing impacts. While one would expect verbatim reuse to offer dramatic reductions in error density, it is not so clear what should be expected from reuse via modification. Understanding the differences in the impact on error density of the various reuse approaches can allow for improved process optimization based on the expected error density profile. For example, it can allow for the development of more accurate models predicting potentially faulty modules in a system. A number of recent studies have shown that product metrics can be used to determine the areas in a program that are at a greater risk of containing a fault (Selby and Porter, 1988; Agresti and Evanco, 1992; Munson and Khoshgoftaar, 1992; Briand et al., 1993). Whether and how the component was reused are also likely to be useful in such models.
2. Are errors in reused components easier to isolate or correct?

Determining the difficulty of error corrections can allow for more accurate allocation of resources and can help to assess the overall effectiveness of reuse in terms of rework effort. Reuse may impact the effort expended to isolate and correct errors, as the level of understanding of a reusable component may be less than that of a new component. On the other hand, reusable components
3. Are the errors typically being introduced or detected at different phases?

Knowledge of when the errors are being introduced and detected enables one to apply verification techniques at the most suitable time. If errors are being introduced early but are not being detected until late, it may be beneficial to more closely examine the verification processes and introduce techniques that can detect the errors at an earlier phase. For example, if a large number of errors are being introduced in the design phase, adding design inspections to the development process may reduce the number of errors impacting later phases. On the other hand, if most errors are being introduced during coding, design inspections may not be as cost effective. In this

case, one may choose not to inspect design but choose to have additional verification effort in the coding phase.

4. Are there different kinds of errors associated with reused units?

Basili and Selby (1987) found that the effectiveness of error detection techniques varies with the type of fault encountered. For example, code reading was found to be the most effective technique for isolating interface errors, while functional testing was found to be more effective at finding logic errors. As such, a priori knowledge of the distribution of the type of errors allows one to select verification techniques most appropriate for that distribution. Suppose two thirds of the errors are interface errors, and one third logic errors. In this case, we would want to be sure to use techniques that are effective in finding interface errors. Given a limited budget for verification and validation, we may choose to expand more resources in code reading and fewer in functional testing. On the other hand, if a different project is much more likely to have logic errors than interface errors, it may be more effective to focus the verification activities on structural testing.

3.2 Data Collected

Several types of data were used in our analyses. The first type of data has to do with the origin of a component—whether it was newly created or reused. At the time of component creation, a form was filled out by the developer, indicating the origin of the component—whether it was to be created new, reused from another component with extensive modification (more than 25% changed), reused with slight modification (less than 25% changed), or reused verbatim (without change). Table 2 provides a summary of the number of components and source statements in each category of component origin. A larger amount of source code was created in the new and reused verbatim categories than in either of the categories of reuse with modification.

Table 2. Profile of Each Class of Component Origin

Component Origin	No. Comp.	KSTMT	Pct. KSTMT
New	1095	44.2	36.5
Extensively Modified	152	8.8	7.2
Slightly Modified	517	21.6	17.8
Reused Verbatim	1495	46.6	38.5
All Components	3259	121.2	100.0

The SEL uses "Change Report Forms" to collect data on changes to components for various reasons, such as error corrections, requirements changes, and planned enhancements. In this analysis, we examined the changes made to correct errors. For each reported error, the form identifies the modules that needed to be changed, the source of error, (requirements, functional specification, design, code, or previous change), the type of error (initialization, computational, data value, logic, internal interface, or external interface), and whether or not the error was one of omission (something was not done) or commission (something was done incorrectly).

Finally, we analyzed the systems with a source code static analysis tool, ASAP, which provided us with a static profile of each compilation unit, including, for example, basic complexity measures such as McCabe's Cyclomatic Complexity and Halstead's Software Science, as well as counts of various types of declarations and statement usage (Doubleday, 1987). ASAP also identifies all with statements, so we were able to develop measures of the external declarations visible to each unit.

4. RESULTS OF THE ANALYSIS

This section presents the major findings from our analysis. We investigated the similarities and differences among the classes of component origin in terms of the nature and impact of the errors in each class. Various statistical methods were used to determine the significance of these differences. Structural characteristics of the components are discussed in 4.1, and the remaining sections describe findings associated with the various dimensions of errors.

4.1 Structural Characteristics

Table 3 shows a collection of measures that characterize the structure of compilation units by class of reuse. Only compilation units that are subprogram bodies were considered, so as not to bias the results with characteristics of instantiations or package specifications. The average number of Ada state-

Table 3. Structural Characteristics of Subprogram Bodies

Component Origin	Ave. No. Statements	Ave. No. Parameters	Ave. No. Withs
New	45.8	2.1	3.5
Extensively Modified	59.9	2.1	7.5
Slightly Modified	41.6	1.9	4.0
Reused Verbatim	24.5	2.8	1.1
All Components	36.8	2.3	2.7

ments provides an indication of the typical size of a component. The number of parameters is a rough measure of the generality of a component. The number of context couples (i.e., the number of “with” statements) provides an indication of the external dependencies of a particular unit.

Since the statement count, number of withs, and number of parameters are not normally distributed (rather, they are non-negative, integer valued variables), nonparametric tests are appropriate to examine differences across the classes of component origin (Gardner and Altman, 1989). As we were investigating differences between classes of origin, we performed the analysis in a pairwise manner. The Mann-Whitney U is a nonparametric version of the two-group unpaired t-test and can be used to test the null hypothesis of independence between the structural characteristic (e.g., component size) and the grouping variable (e.g., component origin) (Gardner and Altman, 1989). For each pair of origins, we used the Mann-Whitney U statistic to test the null hypotheses that the parameterization, context coupling and external dependencies are independent of the two categories of origin.

Table 4 provides a summary of the results of the statistical analysis, showing the p-value from the Mann-Whitney U test for each pair of component origins. The p-value indicates the probability that the observed difference between the classes is due to chance. In terms of size and external dependencies, there is a significant difference between the reused verbatim components and all other classes, as they are smaller in size (fewer source statements) and external dependencies (fewer with statements). The reused verbatim units average 24.5 statements and 1.1 withs per unit, while the new units average 45.8 statements and 3.4 withs per unit. The extensively modified units tend to be the most complex, both in terms of their size and external dependencies, as they average 59.9 statements and 7.5 withs per unit. The p-values of less than 0.01 indicates that there is less than 1% probability that the differences

are due to chance. These results are similar to what was reported by Selby (1988) in his analysis of reuse in a collection of FORTRAN systems. He also found that the slightly modified and verbatim components tend to be simpler than newly created components in terms of size and interaction with other modules, while those with major revisions tend to be more complex.

We did note one result that is in contrast to Selby’s study. He reported that the verbatim reused modules tend to have a smaller interface than newly created units. We observed the opposite—that the verbatim reused modules tend to have more parameters than either the modified or new components. The verbatim reused components averaged 2.8 parameters per unit, versus 1.9 to 2.1 in the new and modified components, and the p-values indicate that these differences are significant. Units that are well parametrized have an increased generality that may allow them to be more readily integrated into new applications. As such, we should see a greater number of parameters in the unchanged modules. This result may be indicative of the approach being taken to reuse in the environment. As previously noted, the Ada approach in this environment was based on the use of well-parameterized generics, while the FORTRAN approach was based on libraries of more specialized functions (Bailey et al., 1993). As such, we might expect a lower level of parameterization in reused FORTRAN modules. Another reason for the difference from Selby’s study may be that his measure of a module’s interface is a sum of counts of the parameters and global references in the module. In the FORTRAN modules that he examined, this sum is likely to be dominated by the count of global references; as such, the variation in the count of subprogram parameters among the classes of reuse can not be observed.

Table 5 shows the profile of the reused components over time, as the projects are listed in chronological order of their development start date. We see an increasing complexity (expressed both in terms of

Table 4. Comparison of Structural Characteristics by Class of Component Origin

	P-Values* from Comparison of:		
	Statements	Parameters	Withs
New—Ext. Mod.	< .0001	.5533	.0250
New—Slt. Mod.	.5609	< .0001	.4073
New—Verbatim	< .0001	< .0001	< .0001
Ext. Mod.—Slt. Mod.	< .0001	.0411	.0172
Ext. Mod.—Verbatim	< .0001	.0004	< .0001
Slt. Mod.—Verbatim	< .0001	.0535	< .0001

* P-Values were obtained from a Mann-Whitney U Test

Table 5. Structural Characteristics in Verbatim Reused Components as Reuse Increases

Project	Ave. No. Statements	Ave. No. Withs	Ave. No. Params.
A	15	0.3	1.9
B	14	0.2	1.8
C	14	0.2	1.8
D	18	0.9	2.7
E	31	1.1	3.0
F	26	1.2	2.1
G	26	1.5	3.1

module size and external dependencies) in the reused components. Also, we see a rise in the number of parameters per subprogram in the verbatim units, suggesting an increasing generality among them. Low level utility functions were the first to be reused, but as the organization gained reuse experience, more and more complex units were reused as well. Thus, while utility functions may be among the best components to initially stock a repository, a reuse process is not limited to them. As this organization gained experience, more and more complex units at higher levels of the application hierarchy were reused.

4.2 Error Density

Table 6 shows the error and defect densities (errors/defects per thousand source statements) observed in each of the four classes of component origin. We use error to refer to a change report in which the reason for the change was attributed to an error correction. A change report can list several components as requiring correction due to a single error. We refer each instance of a component requiring modification due to an error as a defect. As such, there can be several defects associated with a single error. Two measures of error and defect density are shown—the first (labeled “Error Density” and “Defect Density”) includes all errors or defects from unit test through acceptance test, while the second (labeled “S/A Err. Density” and “S/A Def. Density”) only includes those detected in system and acceptance (S/A) test. The first measure can provide an indication of the total density of defects and errors, while the second shows the density that is occurring late in the development life cycle.

There is a clear benefit from reuse in terms of reduced error density when the reuse is verbatim or via slight modification. However, reuse through slight modification only shows about a 59% reduction in total error density, while verbatim reuse results in more than a 90% reduction compared to newly developed code. When we only look at the errors

that are encountered during the system and acceptance test phases, we still see a greater than 90% reduction in defect density in the reused verbatim class (0.7 errors per KSLOC, compared to 8.4 errors per KSLOC, in the new components). The slightly modified components, with 2.5 errors per KSLOC, show a reduction of nearly 70% compared to the new components, with 8.4 errors per KSLOC. In terms of error density, reuse via extensive modification appears to yield no advantage over new code development. Verbatim reuse clearly provides the most significant benefit to the development process in terms of reducing error density, but reuse via slight modification also provides a substantial improvement, one which is even more noticeable later in the development process.

We used the Mann-Whitney U test to obtain a statistical comparison of component defect density by class of component origin. We tested, for each pair of component origin classes, the null hypothesis that there was no significant difference in error density between the classes. A summary of the results is shown in Table 7. This comparison shows a significantly lower error and defect density among the reused verbatim components compared to each of the other classes. The slightly modified components also show significantly lower defect density than the new and extensively modified components. No significant difference was observed between new and extensively modified components.

One question that may arise is whether the differences in component sizes in the different classes of component origins has an effect on the error density. A number of studies have reported higher defect/error densities in smaller components than in larger components (Basili and Perricone, 1984; Shen et al., 1985; Lind and Vairavan, 1989; Möller and Paulish, 1993). One explanation for the higher error density in the small components is that a system composed of small components will have more interfaces than a system composed of large components; and interfaces are often noted as a major source of error in development. We observed

Table 6. Error and Defect Densities in Each Class of Component Origin

Component Origin	No. Comp.	KSTMT	Defect Density	Error Density	S/A Err. Density	S/A Def. Density
New	1095	44.2	24.8	13.0	8.4	18.5
Extensively Modified	152	8.8	19.5	14.0	8.9	13.4
Slightly Modified	517	21.6	10.5	7.4	2.5	5.7
Reused Verbatim	1495	46.6	2.1	1.2	0.7	1.5
All Components	3259	121.2	13.1	7.6	4.4	9.3

Table 7. Comparison of Defect Density by Class of Component Origin

Component Origins	Defect Density	P-Values* from Comparison of:		
		Error Density	S/A Def. Density	S/A Err. Density
New vs Ext. Mod	.5573	.0982	.8168	.1839
New vs Slt. Mod.	< .0001	< .0001	< .0001	< .0001
New vs Verbatim	< .0001	< .0001	< .0001	< .0001
Ext. Mod. vs Slt. Mod.	< .0001	< .0001	< .0001	< .0001
Ext. Mod. vs Verbatim	< .0001	< .0001	< .0001	< .0001
Slt. Mod. vs Verbatim	< .0001	< .0001	< .0001	< .0001

* P-Values were obtained from a Mann-Whitney U Test

a similar effect of higher defect densities in the smaller components of these projects. Table 8 shows the defect densities found in small components (25 or fewer statements) and large components (more than 25 statements), and the p-values from Mann-Whitney U tests for the comparison of defect density by component size (i.e., small vs large). We see that smaller components tend to have higher defect densities, and the p-values indicate that the result is significant. The exception is the class of the verbatim components, where the defect densities are quite low for both small and large components.

Since the verbatim and slightly modified components are smaller than the new and extensively modified components, we would expect that their smaller size would lead to an increased error density, as opposed to the smaller densities that we observed. As such, we concluded that the lower rates of defect density in the verbatim and slightly modified classes do not result from their smaller sizes.

4.3 Error Isolation/Completion Difficulty

Basili and Perricone (1984) in their study of a FORTRAN development project, reported that modified components typically required more correction effort than new components. We see a similar result in the two classes of modified components, and also see

the same pattern occurring in the reused verbatim components. The SEL's change report forms provide categorical data on the effort to isolate and complete each error correction, with the categories defined as less than 1 hour, 1 hour to 1 day, 1 to 3 days, and more than 3 days. Table 9 shows the percentage of errors in each class of reuse that were categorized in the top two classes, i.e., requiring more than one day to isolate (labeled "Pct. Diff. Isol.") and more than one day complete (labeled "Pct. Diff. Comp."). The last column of this table shows the relative rework effort, a computed approximation of relative effort (staff-hours per KSTMT) in isolating and correcting errors.

We do not see much variation in the effort to isolate an error, as the percentage of difficult-to-isolate errors ranges from 12.4% for new components to 14.5% for the extensively modified components. However, we do see a greater difference in the difficult-to-complete errors. The reused verbatim components had the highest percentage of errors requiring more than one day to complete an error correction, and the new components had the lowest percentage, while the modified components fell in between. One explanation for this effect is that the developers have a greater familiarity with the newly created components, so less time is needed to understand the components that must be changed. Another explanation is that the majority of the

Table 8. Relationship of Defect Density and Component Size

Component Origin	Small		Large		P-Value* Small vs Large
	No. Comp.	Def. Dens.	No. Comp.	Def. Dens.	
New	638	49.8	457	19.8	.0003
Extensively Modified	67	35.7	85	17.7	.0660
Slightly Modified	283	26.5	234	7.4	.0036
Reused Verbatim	952	2.3	543	2.0	.1982
All Components	1940	22.6	1319	10.9	< .0001

* P-Values were obtained from a Mann-Whitney U Test

Table 9. Difficulty in Error Isolation/Correction

Component Origin	No. Errors.	Pct. Diff. Isolation	Pct. Diff. Completion	Rel. Rework Effort
New	574	12.4	10.1	118.3
Extensively Modified	124	14.5	17.7	157.4
Slightly Modified	160	13.8	13.1	76.8
Reused Verbatim	58	14.3	22.4	14.7
All Components	916	13.2	12.6	73.9

“easy” errors had previously been removed from the reused component, leaving only the more difficult ones.

To test whether the differences in the effort distributions were significant, we used a chi-square test, since both measures (component origin and effort) are categorical (Gardner and Altman, 1989). As with the previous analyses, we used the test in a pairwise manner, to test (for each pair of origins) the null hypothesis of independence between component origin and error isolation/correction effort. The results of this analysis are summarized in the second and third columns of Table 10, which shows the p-values from the chi-square test, indicating the probability that the differences between classes of component origin in the distribution of isolation and completion effort are due to chance.

We do see significant differences in the effort distributions between classes of component origins. In terms of isolation effort, all pairs showed a significant difference, except for the pair of new and slightly modified components. For the effort to complete an error correction, all pairs showed a significant difference, except for the pair of new and extensively modified components.

To determine whether the increased error correction cost in the reused components outweighs benefit of their having fewer errors, we computed a rough measure of the amount of error rework expended in

each class. Unfortunately, our data for effort spent in error correction and isolation is categorical, so we approximated the true effort simply by the midpoint of the category (μ). Rework effort was then computed as the sum of this approximation over all errors. Our relative rework effort measure (RR) was computed by dividing rework effort by the number of statements (S), i.e.,

$$RR = \frac{\sum_{i=1}^n \mu(e_i)}{S}$$

Again, we used the Mann-Whitney U test to determine whether there is a significant difference in the relative rework effort among the four classes of component origin. As shown in the last column of Table 10, the tests found significant differences between the classes with one exception—there is not a significant difference in relative rework effort between new and extensively modified components. For all other pairs, the result was significant at the 0.01 level. As shown in Table 9, reuse via slight modification shows a 35% reduction in rework cost over newly created components, while verbatim reuse provides an 88% reduction. For these modes of reuse, the benefit of fewer errors clearly outweighs the cost of more difficult error correction. This measure of benefit is somewhat conservative, as it does not account for the expected reduction in component creation cost, or for the impact of errors as “obstacles” in the development process (e.g., the cost of delays due to effort spent correcting errors). As such, we expect these modes of reuse to yield an even greater improvement over new development. This shows that there is a shift in costs of reuse compared to traditional development, with the reuse-oriented development showing less development effort and fewer, but more costly, errors.

4.4 Source of Errors

Understanding the activity in which the error is introduced allows for corrective action to be applied at the appropriate time. The SEL change report forms indicate the “source” of the error, which can be requirements, functional specification, design, code, or a previous change. Table 11 shows, for each class of component origin, the percentage of errors from each error source (where the error was introduced). Across all classes, “code” is the most common error source; however, there do appear to be some differences.

One result that appears interesting is that errors associated with requirements and functional specification occur at a slightly higher rate in new compo-

Table 10. Comparison of Error Isolation/Correction Effort

	P-Values from Comparison of:		
	Isolation* Effort	Completion* Effort	Rel. Rework** Effort
New—Ext. Mod.	< .0001	.5533	.9421
New—Slt. Mod.	.5609	< .0001	< .0001
New—Verbatim	< .0001	< .0001	< .0001
Ext. Mod.—Slt. Mod.	< .0001	.0411	< .0001
Ext.—Mod. Verbatim	< .0001	.0004	< .0001
Slt. Mod.—Verbatim	< .0001	.0535	< .0001

* P-Values were obtained from a Chi Square Test

** P-Values were obtained from a Mann-Whitney U Test

Table 11. Percentage of Errors in Each Class of Error Source

Component Origin	Rqmts. or Fun. Spec.	Design	Code	Previous Change	Any Error
New	7.3	16.8	68.1	7.8	100
Extensively Modified	5.6	20.2	59.7	14.5	100
Slightly Modified	4.4	26.9	60.1	10.6	100
Reused Verbatim	3.4	3.4	74.1	19.0	100
All Components	5.7	18.2	66.1	10.0	100

nents than in reused components. The Basili-Pericone (1984) study reported the opposite effect of reuse on the specification errors. They found that modified modules had a higher proportion of specification errors than did the new modules and explained the result by suggesting that the specification was not well enough or appropriately defined to be used in different contexts. A similar result was reported by Endres (1975). A difference from the environments examined in those studies is that reuse has been well planned for in the SEL environment. The organization is not structured as a pure "component factory" as described in Section 3, but it is moving in this direction. As such, the architecture, design, and specifications have improved in this environment to better allow and encourage reuse. This result suggests that the reused functionality is more likely to be well specified. This is not surprising, since the reused components have been specified previously, with the expectation that they would be reused. As such, any specification errors are more likely to affect new rather than reused components. The result also indicates that reuse, whether formal or informal, is occurring in this environment at a higher level than simply code.

The verbatim components have the highest percentage of errors associated with a "previous change" (nearly 20%, which is an increase over the 8 to 14% observed in the other classes.) This may be due to the fact that the previous uses of these components have already found most of their errors, and thus, the increased rate of error "reintroduction" may be explained more as a decrease in the other types of errors.

Another item of interest is the increased percentage of design errors in the modified components. We see a different effect in the verbatim components, where almost all errors have their source as being "code" or a "previous change." These results suggest that there is increased difficulty in designing an adaptation of an existing component to a new role. This is more difficult because the reuser must be concerned with two pieces of information: the

intended function and the existing function. In creating a new component, one only needs to be concerned with the intended function, and for a verbatim component, with the existing function. For the modified components, a misunderstanding of the existing function can result in an error in the component modification, and that error is likely to be attributed to the design of the modification.

The statistical comparison between classes of component origins of the distribution of errors across error sources is shown in Table 12. Again, a chi-square test was used to test the null hypothesis that there was no significant difference in the distribution of error sources across component origins. We see no significant differences between new and extensively modified components, or between extensively and slightly modified components. The differences between all other pairs are statistically significant. These results support the observations of increased design errors in modified components, increased requirements and specification errors in new components, and increased errors due to a previous change in the unchanged components.

4.5 Time of Error Detection

Errors detected late in the development life cycle can have a much greater cost than those detected early. Table 13 shows, by class of component origin, the percentage of errors that "escape" unit test, and are detected in the system or acceptance test phases. Clearly, low rates of error slippage are desired. Three columns of error slippage rates are shown, with the first showing the rate for all development errors ("Pct. All Errors"), and the second and third columns showing the rates for the errors that were more difficult to isolate and complete ("Pct Diff. Isolation" and "Pct Diff. Completion", resp.) The difficult errors escape at an increased rate, supporting the notion of a tendency towards higher costs associated with errors late in the life cycle.

Table 12. Comparison of Error Source by Class of Reuse

Component Origins	P-Values* from comparison of error sources
New—Ext. Mod.	.2126
New—Slt. Mod.	.0036
New—Verbatim	.0073
Ext. Mod.—Slt. Mod.	.1476
Ext. Mod.—Verbatim	.0223
Slt. Mod.—Verbatim	.0013

* P-Values were obtained from a Chi-Square Test

Table 13. Percentage of Errors That Escape Unit Test

Component Origin	Pct. All Errors.	Pct. Diff. Isolation	Pct. Diff. Completion
New	69	86	80
Extensively Modified	66	81	87
Slightly Modified	43	74	58
Reused Verbatim	62	100	100
All Components	64	84	78

Across all errors, we see little difference between the classes of new, extensively modified, and reused verbatim components, as nearly two thirds of the errors in these classes escaped unit test. This is significantly higher than what we observe in the slightly modified components, where only 43% escaped unit test. It appears that the nature of the changes being made to these components lend themselves well to detection by unit-level verification processes.

Of the errors which required the greatest isolation effort (those taking more than one day to isolate), there is not much difference among the classes—a relative high percentage of these errors escape in all classes, and we saw no significant differences across the origins. There is a significant reduction in the slightly modified class compared to the other modes of reuse, in the percentage of difficult-to-complete errors that escape unit test, as only 58% of these errors escape unit test, compared to 87% and 100% in the extensively modified and verbatim classes. This suggests that the verification process is more effective in the early elimination of difficult errors for the slightly modified components than for other modes of reuse.

A summary of the chi-square test comparing the error slippage across component origins is shown in Table 14, with the columns showing a breakout of all errors, the difficult isolation errors (more than 1 day of isolation effort), and the difficult completion errors (more than 1 day of completion effort). The results that are significant with p-values of less than

Table 14. Comparison of Defect Slippage

Component Origins	P-Value* for Comparison of:		
	All Errors	Difficult Isolation	Difficult Completion
New—Ext. Mod.	.8303	.4728	.3620
New—Slt. Mod.	< .0001	.1756	.1243
New—Verbatim	.4561	.3166	.1063
Ext. Mod.—Slt. Mod.	.0003	.1302	.0508
Ext. Mod.—Verbatim	.6149	.4966	.2629
Slt. Mod.—Verbatim	.0199	.1396	.0195

* P-Values were obtained from a Chi Square Test

.10 involve the slightly modified components, supporting the observation of a significantly lower error slippage rate in the slightly modified components.

4.6 Type of Errors

The SEL also categorizes errors by their type, with the categories being logic, computational, internal interface, external interface, data/value, and initialization. We grouped these into three classes as follows: procedural errors are those that were classified as either a computational or a logic error; interface errors are those that were classified as either an internal or external interface error; and data errors are those that were classified as either an initialization or a data value error. Table 15 shows the percentage of errors that were classified in each of the three classes: procedural, interface, and data. Again, a chi-square test was used to test whether the differences in the distribution of error types were significant across classes of component origins. The results of this test are summarized in Table 16.

We see a significant difference in the distribution of error types in the slightly modified components, as they have a much higher frequency of interface errors than any other class. This suggests that the nature of the modifications is likely to be associated with the interface. We also see that the new components are more likely to have data errors than the reused components. However, the p-values from the comparison of the new vs verbatim and new vs extensively modified classes indicate that the differences may not be significant. Basili and Perricone (1984) found the opposite effect, namely, that the modified components had a greater percentage of data errors than did the new components. Our results suggest that a different approach has been taken toward reuse. In the FORTRAN project studied by Basili and Perricone, the approach may have been to tailor data values and initialization to adapt the component to the new application. The approach taken in the Ada environment is to create

Table 15. Percent of Errors of Each Type by Class of Component Origin

Component Origin	Procedural	Interface	Data	All
New	41.2	14.1	44.6	100
Extensively Modified	47.6	17.7	34.7	100
Slightly Modified	31.8	31.2	36.9	100
Reused Verbatim	48.2	12.1	39.7	100
All Components	40.9	17.5	41.6	100

Table 16. Comparison of Error Type by Class of Reuse

Component Origins	P-Value* for comparison of error type
New—Ext. Mod.	.1221
New—Slt. Mod.	< .0001
New—Verbatim	.5878
Ext. Mod.—Slt. Mod.	.0084
Ext. Mod.—Verbatim	.5850
Slt. Mod.—Verbatim	.0099

* P-Values were obtained from a Chi-Square Test

generalized modules that can be parameterized to create instances suitable for the new application. As such, one might expect fewer data errors in reused components in the Ada environment.

5. CONCLUSIONS

In this analysis, we observed clear benefits from reuse—for example, reduced error density. We found that verbatim reuse provides a substantial improvement in error density (more than a 90% reduction) compared to new development. The other modes of reuse did not approach this level of improvement. Reuse via slight modification offered a 50% reduction in error density compared to new development, but the improvement with this mode of reuse was greater in errors detected late in development (a 70% reduction).

We observed a shift in costs of reuse-oriented development, with the reuse offering fewer, but more difficult errors. The effect of increased difficulty in error correction was apparent across the three modes of reuse, although it was less evident in the slightly modified components. In both the verbatim and slightly modified classes of reuse, the relative amount of rework was less than in new code. This suggests that while there is a cost of increased correction effort per error associated with such reuse, the cost is outweighed by the benefit of the reduced number of errors. Coupled with the reduction in development effort, these modes of reuse appear to offer a substantial benefit to development.

Reuse via extensive modification does not provide the reduction in error density that the other modes of reuse yield, and it also results in errors that typically were more difficult to isolate and correct than the errors in newly developed code. In terms of the rework due to the errors in these components, it appears that this mode of development is more costly than new development. However, extensive modification may offer savings in development effort that outweigh the increased cost of rework. This

remains an issue for further study.

A different profile of errors was observed for different modes of reuse. For example, a greater percentage of design errors was observed in the modified components. The observed increase in design errors may be due to errors in the additional activities of understanding the function and implementation of the component to be modified, as well as due to the fact that less code was being written. Such information can be used to help in selecting appropriate verification methods for projects where there is significant reuse via modification. One may want to increase the effort in design reviews on such projects, while on projects dominated by new development, code reviews may receive more emphasis. This finding also suggests that one might want to investigate techniques to better describe the components stored in the experience base so that the likelihood of a misunderstanding of the function and implementation is lessened.

The experience with reuse in an organization and the approach taken toward reuse are likely to influence the nature of errors. In this study of an organization well experienced with reuse, we observe a number of effects that differed with findings from other studies of environments where reuse was more ad hoc. The reused components appear to be simpler, have fewer dependencies, and be more parameterized than new components. However, as this organization gained reuse experience, the distinction became less apparent—more and more complex components, at higher levels in the application hierarchy were reused. As an organization moves toward a reuse-oriented development approach, it must evolve its practices to accommodate the new effects of reuse. Error analysis is a useful mechanism to provide insight into the benefits and difficulties of reuse in software development.

ACKNOWLEDGMENT

This work was supported in part by the National Aeronautics and Space Administration grant NSG-5123 and the Center for Advanced Technology in Telecommunication (CATT), Brooklyn, NY.

REFERENCES

- Agresti, W. W., and Evanco, W. M., Projecting Software Defects from Analyzing Ada Designs. *IEEE Transactions on Software Engineering*, 18(11) (November 1992).
- Agresti, W., and McGarry, F., The Minnowbrook workshop on software reuse: A summary report. In: (W. Tracz, ed.), *Software Reuse: Emerging Technology*. IEEE Computer Society Press, 1987.
- Bailey, J., Waligora, S., and Stark, M., Impact of Ada in

- the flight dynamics divisions: Excitement and frustration. In: *Proceedings of the 18th Annual Software Engineering Workshop*. NASA/GSFC, December 1993.
- Basili, V. R., and Perricone, B. T., Software Errors and Complexity: An Empirical Investigation. *Communications of the ACM*, 27(1) (January 1984).
- Basili, V., and Rombach, D., The TAME Project: Towards Improvement-Oriented Software Environments. *IEEE Transactions on Software Engineering*, 14(6) (June 1988).
- Basili, V. R., and Rombach, H. D., Support for Comprehensive Reuse. *Software Engineering Journal*, 6(5) (September 1991).
- Basili, V. R., and Selby, R. W., Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, 13(12) (December 1987).
- Basili, V. R., Caldiera, G., and Cantone, G., A Reference Architecture for the Component Factory. *ACM Transactions on Software Engineering and Methodology*, 1(1) (January 1992).
- Boehm, B. W., and Papaccio, P. N., Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering*, 14(10) (October 1988).
- Briand, L. C., Thomas, W. M., and Hetmanski, C. J., Modeling and managing risk early in software development. In: *Proceedings of the Fifteenth International Conference on Software Engineering*, May 1993.
- Brooks, F. P., No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4) (April 1987).
- Caldiera, G., and Basili, V. R., Identifying and Qualifying Reusable Software Components. *IEEE Computer*, 24(2) (February 1991).
- Doubleday, D., ASAP: An Ada Static Source Code Analyzer Program. Technical Report CS-TR-1897, University of Maryland, May 1987.
- Dunn, M., and Knight, J., Automating the detection of reusable parts in existing software. In: *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, Maryland, May 1993.
- Ebel, N., and Genillard, C., The reusability of Ada software components. In: (R. Gautier and P. Wallis, eds.), *Software Reuse with Ada*. Peter Peregrinus Ltd., 1990.
- Endres, A., An analysis of errors and their causes in system programs. In: *Proceedings of the International Conference on Software Engineering*, April 1975.
- Gardner, M., and Altman, D., *Statistics with Confidence: Confidence Intervals and Statistical Guidelines*, British Medical Journal, London, 1989.
- Gargaro, A., and Pappas, T., Reusability Issues and Ada. *IEEE Software* (July 1987).
- Ichbiah, J., *The Rationale for the Ada Programming Language*, Cambridge University Press, 1985.
- Jones, T. C., Reusability in Programming: A Survey of the State of the Art. *IEEE Transactions on Software Engineering*, SE-10(5) (September 1984).
- Kester, R., SEL Ada Reuse Analysis and Representations. In: *Proceedings of the 15th Annual GSFC Software Engineering Workshop*. NASA/GSFC, November 1990.
- Lanergan, R., and Grasso, C. Software Engineering with Reusable Designs and Code. *IEEE Transactions on Software Engineering*, SE-10(5) (September 1984).
- Lind, R., and Vairavan, K., An Experimental Investigation of Software Metrics and their Relationship to Software Development Effort. *IEEE Transactions on Software Engineering*, 15(5) (May 1989).
- Möller, K., and Paulish, D., An Empirical investigation of software fault distribution. In: *Proceedings of the First International Software Metrics Symposium*, Baltimore, Maryland, May 1993.
- Munson, J., and Khoshgoftaar, T., The Detection of Fault-Prone Programs. *IEEE Transactions on Software Engineering*, 18(5) (May 1992).
- Rombach, H. D., Software Reuse: A Key to the Maintenance Problem. *Information and Software Technology*, 33(1) (January/February 1991).
- An Overview of the Software Engineering Laboratory. Technical Report SEL-94-005, Software Engineering Laboratory, NASA Goddard Space Flight Center, December 1994.
- Selby, R. W., and Porter, A. A., Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis. *IEEE Transactions on Software Engineering*, 14(11) (November 1988).
- Selby, R., Empirically analyzing software reuse in a production environment. In: (W. Tracz, ed.), *Software Reuse: Emerging Technology*. IEEE Computer Society Press, 1988.
- Shen, V., Yu, T., Thebaut, S., and Paulsen, L., Identifying Error-Prone Software—An Empirical Study. *IEEE Transactions on Software Engineering*, SE-11(4) (April 1985).
- Stark, M., Impacts of object-oriented technologies: Seven years of SEL studies. In: *Proceedings of Eighth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, October 1993.
- Thomas, W. M., Delis, A., and Basili, V. R., An Evaluation of Ada source code reuse. In: (J. van Katwijk, ed.), *Ada: Moving Towards 2000 (Proceedings of the Ada-Europe International Conference)*, Zandvoort, The Netherlands, June 1992. Springer-Verlag.
- Wolf, A., Clarke, L., and Wileden, J., Ada-Based Support for Programming in the Large. *IEEE Software* (March 1985).