AN ANALYSIS OF ERRORS IN A REUSE-ORIENTED DEVELOPMENT ENVIRONMENT*

William M. Thomas	Alex Delis	Victor R. Basili
Dept. of Computer Science	School of Information Systems	Dept. of Computer Science
University of Maryland	Queensland Univ. of Technology	University of Maryland
College Park, MD 20742	Brisbane, QLD 4001, Australia	College Park, MD 20742

Abstract

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 paper, 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, when errors are introduced, and the effect that the errors have on the development process.

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 [AM87, Bro87, BP88]. 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 present a framework of comprehensive support for reuse, including organizational and methodological properties necessary to maximize the benefit of reuse [BR91]. 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

*This was supported in part by the National Aeronautics and Space Administration grant NSG–5123.

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 [TDB92]. 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 [CB91, DK93]. One factor to be considered is the structure of the candidate reusable component. Selby investigated various characteristics of new versus reused code in a large collection of FORTRAN projects [Sel88]. Basili and Perricone analyzed tradeoffs between creating a component from scratch versus modifying an existing component [BP84]. This work extends these studies by investigating the nature of errors occurring in a reuse oriented development environment, and drawing conclusions as to 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 some similarites and some differences with the findings of other investigations into component reuse.

This paper 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 estimates that only 15 percent of the developed software is unique to the applications for which it was developed [Jon84]. Reduced development cost is not the only benefit of reuse–in fact, the greatest benefit from reuse may be its impact on maintenance [LG84, Rom91]. The potential for substantial savings from reuse clearly exists. Unfortunately, achieving high levels of reuse still remains an difficult 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 differentiating verbatim reuse and reuse via modification is trivial, distinguishing between slight modification and extensive modification is more difficult. Our intent is to distinguish between cases where a component is left essentially intact, but needs some small change for the new application, and cases where a component is significantly altered for its new use. The three types of reuse, and a 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, since 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 to 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; but, because the modified components may be adapted to better match the application, the integration is perhaps not 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. However, 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 [Ich85, WCW85, GP87, EG90]. 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 [BWS93]. 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 percent verbatim reuse), while the FORTRAN approach showed less than 10 percent verbatim reuse [BWS93]. 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.

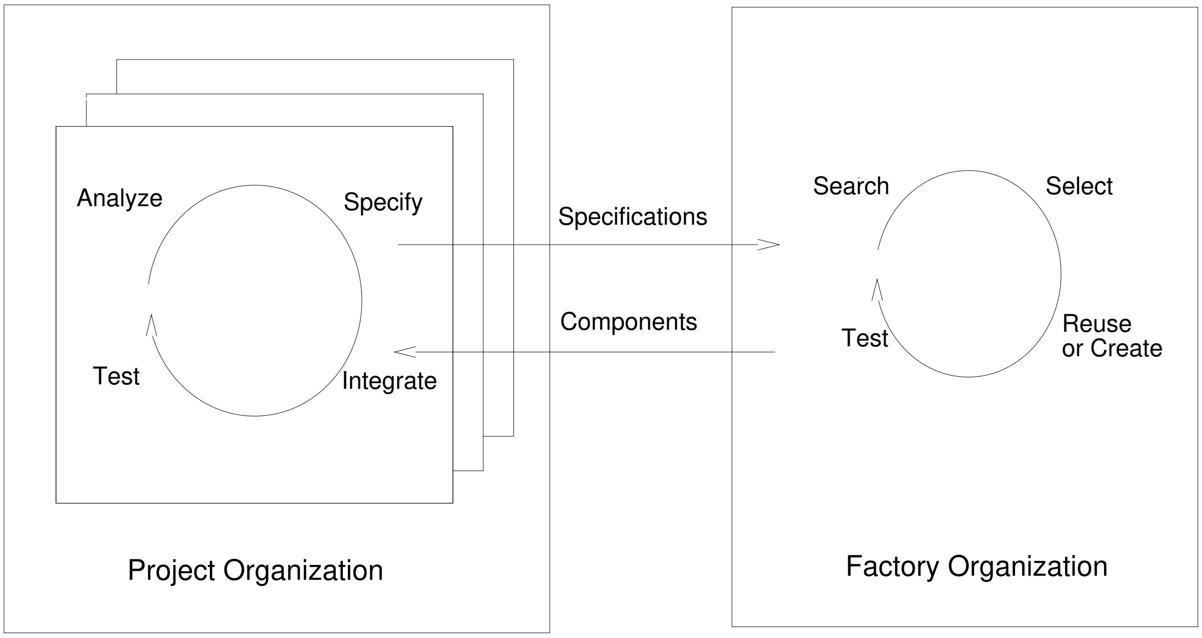

Figure 1: Interaction of a Project Organization with the Component Factory

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 [BCC92]. 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 utilized, such as that proposed in the TAME project [BR88], 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.

3 Using Error Analysis to Optimize the Development Process

The Quality Improvement Paradigm provides a framework to build a continually improving organization relative to its evolving set of goals [Bas85, BR88]. The QIP consists of six steps:

1. **Characterize** the current project and environment.

2. **Set Goals** for project performance and improvement.

3. **Choose processes**, as well as models and metrics, appropriate for the project.

4. **Execute** the processes, and collect the prescribed data, and provide real-time feedback for corrective action.

5. **Analyze** the data to evaluate current practices and make recommendations for future improvement.

6. **Package** the experience in a form suitable for reuse on future projects.

The first two steps deal with determining the nature of the project, including goals for performance and improvement. Based on the characterization and goals, the third step selects the most suitable processes for the project; establishes the measurement plan, including choosing appropriate models and metrics, and sets up the mechanism for real-time feedback as the project progresses. The fourth step starts the selected processes, collects and the data as prescribed by the measurement plan, and uses the selected models and metrics to provide feedback to the development organization. The fifth and sixth steps occur off-line, as the data is analyzed and packaged into the experience base for use in other projects.

Examining the various dimensions of errors in an organization can yield important lessons learned that may be used to improve software development. 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. Thus error analysis can be associated with either of the two feedback loops in the model, the project loop, occurring in step 4, in which the results are in real-time provided back to the project, or the corporate loop, in steps 5 and 6, in which results are made available for subsequent projects in the organization. Our focus in this paper is on the corporate loop; i.e., the analysis and packaging steps for subsequent development, from the perspective of reuse-oriented software development.

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 [AE92, SP88, BBH93, BTH93, MK92]. These studies indicate that models can be developed to isolate faulty components in a system based on characteristics of the components and their environment. 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.

Basili and Selby found that the effectiveness of error detection techniques varies with the type of fault encountered [BS87]. 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 the 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 expend 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.

Knowledge of when the errors are being introduced enables one to apply verification techniques at the most suitable time. 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.

The QIP can be used to take advantage of such knowledge. To incorporate this reuse information into the development process, we can develop a mapping to the QIP. The first step of the QIP, characterize the project, can be tailored to include determining the amount and type of reuse expected on the project. The second step, select appropriate models, can include selecting models of expected error profiles based on the characterization of reuse. The third step is to select the appropriate processes. Here, one can choose the processes expected to be most effective for the expected error distribution. The fourth and fifth steps are to execute the processes, collect data, and feedback the results. This can be seen as measuring the actual reuse profile, and measuring the effectiveness of the error mitigation strategies, and making a determination of whether to modify the selected processes based on the new information. For example, if the actual reuse profile is very different from original expectations, one should attempt to understand the factors that led to the difference, and, if appropriate, develop a new projection of the expected error profile.

4 Description of the Analysis

Since its origin, The NASA/GSFC SEL has collected a wealth of data from their software development [SEL94]. Selby performed a study on the characteristics of reused components on a collection of FORTRAN projects from this environment [Sel88], in which the level of reuse averaged 32 percent. 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 [Sel88] has been achieved more recently in this environment [Kes90]. The high levels of reuse have been attributed in part to the Ada language constructs and object-oriented methods [Kes90, Sta93, BWS93]. More recently,

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

Table 1: Overview of the Examined Projects

however, even the FORTRAN systems have been showing high levels of reuse, although the nature of the reuse is different than reuse in the Ada development environment.

We analyzed a collection of seven medium-scale Ada projects from a narrow domain, as all are simulators which were developed at the NASA/GSFC Flight Dynamics Division. An overview of the 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. Reuse ranged from 4 to 89 percent (verbatim), and from 31 to 100 percent (verbatim and with modification).

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 [Sta93], 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 organization is the separate organization that produces or releases all reusable software products [BCC92]. 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 percent on an early simulator, to only 18 percent on one of the more recent simulators [Sta93]. 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.

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?
2. Are errors in reused units easier to isolate or correct?
3. Are the errors typically being introduced at different phases?
4. Are errors associated with reused units detected earlier in the lifecycle?

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

Table 2: Profile of each class of component origin

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

6. Are there structural differences between new and reused units?

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 percent changed), reused with slight modification (less than 25 percent 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.

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 the error, (requirements, functional specification, design, code, or previous change), the type of the 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 [Dou87], 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. ASAP also identifies all **with** statements, so we were able to develop measures of the external declarations visible to each unit.

5 Results of the Analysis

This section presents the major findings from our analysis. We used non-parametric statistical methods to test the hypotheses there were significant differences among the classes

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

Table 3: Structural Characteristics of Subprogram Bodies

of component origin in terms of the the nature and impact of the errors in each class. Structural characteristics of the components are discussed in 5.1, and the remaining sections describe findings associated with with the various dimensions of errors.

5.1 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 statements 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.

What we see is that the reused verbatim components are simpler in terms of their size and external dependencies, as evidenced by the number of source statements and 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, as they average 59.9 statements and 7.5 withs per unit. The slightly modified units tend to be slightly smaller than the new units, but with roughly the same number of external dependencies. It is interesting to note that the extensively modified components are the most complex, both in terms of their size and external complexity. These results are similar to what was reported by Selby in his analysis of reuse in a collection of FORTRAN systems–the reused components tend to be simpler than newly created components in terms of size and interaction with other modules [Sel88]. This additional complexity may result in an increase in difficulty associated with these components in terms or their error density and error correction effort.

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. This difference is significant at the 0.01 level (i.e., there is less than a one percent chance that there actually is no difference

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

Table 4: Structural Characteristics in Verbatim Reused Components as Reuse Increases

between the classes). Units that are more highly parameterized have an increased generality that may allow them to be more readily integrated into new applications. As such, we should expect to see a greater number of parameters in the unchanged modules. This difference 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 [BWS93]. 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 4 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 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 an organization gains experience, more and more complex units, at higher levels of the application hierarchy may be reused.

5.2 Error Density

Table 5 shows the error and defect densities (errors/defect 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*.

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

Table 5: Error densities in each class of component origin

As such, there can be several defects associated with a single error. Two measures of error density are shown–the first includes all errors from unit test through acceptance test, while the second only includes those detected in system and acceptance test. The first measure can provide an indication of the total amount of rework, while the second shows the amount that is occurring late in the development life-cycle. The measure of defect density shown in the table includes defects from unit through acceptance test.

We used a non-parametric test to obtain a statistical comparison of component error density by class of component origin. This comparison shows a significantly lower error density among the reused verbatim components compared to each of the other classes. Similarly, there is a significant difference between the slightly modified components, and the new and extensively modified components. No significant difference was observed between new and extensively modified components.

In terms of error density, reuse via extensive modification appears to yield no advantage over new code development. There is a 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 50 percent reduction in total error density, while verbatim reuse results in more than a 90 percent reduction. When we only look at the errors that are encountered during the system and acceptance test phases, we still see a greater than 90 percent 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 percent compared to the new components, with 8.4 errors per KSLOC. 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 in the test phases.

A number of studies have found higher defect/error densities in smaller components than in larger components [BP84, SYTP85, LV89, MP93]. As shown in table 6, our data supports their findings. Small components (25 or less statements) have defect density more than twice that of the larger components (more than 25 statements), and this difference is highly significant. The only class of reuse where we saw no significant difference was the reused verbatim components, as they have the same defect density regardless of size. The defect density in the small components was more than twice that of the larger components in the new and extensively modified classes, and nearly four times greater in the slightly modified

Component Origin	Small		Large	
	No. Comp.	Def. Dens.	No. Comp.	Def. Dens.
New	638	49.8	457	19.8
Extensively Modified	67	35.7	85	17.7
Slightly Modified	283	26.5	234	7.4
Reused Verbatim	952	2.3	543	2.0
All Components	1940	22.6	1319	10.9

Table 6: Relationship of defect density and component size

class. One explanation for 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 frequently noted as a major source of error in development.

5.3 Error Isolation/Completion Difficulty

Basili and Perricone, in their study of a FORTRAN development project, reported that modified components typically required more correction effort than new components [BP84]. We see a similar result in the two classes of modified components, and also see the same pattern occurring in the reused verbatim components. Table 7 shows the percentage of errors in each class of reuse that were categorized as difficult to isolate or difficult to complete (defined as more than one day to isolate or complete, resp.), and the relative rework effort, a crude approximation of relative effort (staff-hours per KSTMT) in isolating and correcting these errors. In terms of effort to isolate, we see little difference among the classes of component origin. Newly created components had the smallest percentage of difficult-to-isolate errors, but it was not significantly different from any of the classes of reused components. This result is not surprising, as the isolation activity is associated more with understanding the intended functions rather than with their implementation. As such, the origin of the components may not have as great an impact on isolation effort as it will have on completion effort.

We do see an increase in the effort to complete an error in reused components relative to new components. The new components had the lowest percentage of errors requiring more than 1 day to complete a change and the reused verbatim components had the highest percentage, while the modified components fell in between. The difference between the new and the reused verbatim components is significant at the 0.05 level. 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 "easy" errors had previously been removed from the reused component, leaving only the more difficult ones.

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

Component Origin	KSTMT	No. Errors.	Pct. Diff. Isolation	Pct. Diff. Completion	Rel. Rework Effort
New	44.2	574	12.4	10.1	118.3
Extensively Modified	8.8	124	14.5	17.7	157.4
Slightly Modified	21.6	160	13.8	13.1	76.8
Reused Verbatim	46.6	58	14.3	22.4	14.7
All Components	121.2	916	13.2	12.6	73.9

Table 7: Difficulty in error isolation/correction

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

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

Again, we used a non-parametric test to determine whether there is a significant difference in the relative rework effort among the four classes of component origin. The tests found a significant difference among the classes with one exception. When comparing the extensively modified components and the new components we found the level of significance to be only 0.18. There may be an increase in the rework cost of extensively modified components, however, our data does not confirm this. In any event, it is not clear whether such an increase in rework cost would be offset by the expected benefit of reduced component creation cost.

For all other pairs, the result was significant at the 0.01 level. Reuse via slight modification shows a 35 percent reduction in rework cost over newly created components, while verbatim reuse provides an 88 percent 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.

5.4 Source of Errors

Understanding the activity in which the error is introduced allows for corrective action to be applied at the appropriate time. Table 8 shows, for each class of component origin, the percentage of errors from each error source (when the error was introduced). Across all

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

Table 8: Percentage of errors in each class of error source by class of reuse

classes, coding errors are the most common error; however, errors associated with requirements, functional specification and design occur at a slightly higher rate in new components than in reused components. The Basili-Perricone study reported the opposite effect of reuse on the specification errors [BP84]. 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 [End75]. A difference from the environments examined in those studies is that reuse has been well planned for in this environment. The organization is not structured as a pure "component factory" as described in section 3, but it is moving in that 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 components 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.

A second item of interest is the increased percentage of design errors in the modified components. This suggests 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. A misunderstanding of the existing function can result in an error, and that error is likely to be attributed to the design.

5.5 Time of Error Detection

Errors detected late in the development life-cycle can have a much greater cost than those detected early. Table 9 shows, by class of component origin, the percentage of all errors and the more difficult errors that escape unit test. 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 observed in the slightly modified components, where only 43 percent escaped unit

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

Table 9: Percentage of errors that escape unit test

Component Origin	Error of Omission	Both	Error of Comission	Any
New	35.4	28.6	36.0	100
Extensively Modified	40.3	29.4	30.3	100
Slightly Modified	39.6	20.8	39.6	100
Reused Verbatim	26.3	26.3	47.3	100
All Components	36.2	27.2	36.6	100

Table 10: Percentage of errors of omission and commission

test.

Of the difficult isolation errors (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. However, again, the slightly modified components do show the lowest percentage. There is a significant reduction in the slightly modified class in the percentage of difficult-to-complete errors that escape unit test, as only 58 percent of these errors escape unit test, compared to 80 to 100 percent in the other classes. This suggests that the verification process is more effective in eliminating the difficult errors for the slightly modified components than for other modes of component creation.

5.6 Nature of the Errors

Table 10 shows the percentage of errors that were classified as one of omission, commission, or both. An error associated with a component that was reused verbatim is more likely to be error of commission, and less likely to be one of omission. This suggests that the reused component was typically complete, i.e., it contained the necessary functionality, but at times was in error.

Extensively modified components are more likely to have errors of omission than errors of commission. This may be an indication of the greater complexity of these components. Another possible explanation is that in the development of these components, the intended

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 11: Percent of errors of each type by class of component origin

function was not so clear, resulting in necessary parts being omitted. Additional review of the completeness of the design of these components may be a means for removing these errors at an earlier stage.

New and extensively modified components have a higher rate of errors that are classified as both omission and commission than do the slightly modified or reused verbatim components. This may be due to the nature of new development–it is more likely to result in a complex error.

5.7 Type of Errors

Table 11 shows the percentage of errors that were classified in each of the three classes: procedural, interface, and data. 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.

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. Basili and Perricone found the opposite effect, namely, that the modified components had a greater percentage of data errors than did the new components. These 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 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.

6 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 percent reduction) compared to new development. The other modes of reuse did not approach this level of improvement. Reuse via slight modification offered a 50 percent 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 percent 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 were 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 observed in the organization. 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 not planned for to such an extent. 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. In the context of the QIP, error analysis can be a useful mechanism to provide insight into the benefits and difficulties of reuse in software development.

References

[AE92] W. W. Agresti and W. M. Evanco. Projecting software defects from analyzing Ada designs. *IEEE Transactions on Software Engineering*, 18(11), November 1992.

[AM87] W. Agresti and F. McGarry. The Minnowbrook workshop on software reuse: A summary report. In W. Tracz, editor, *Software Reuse: Emerging Technology.* IEEE Computer Society Press, 1987.

[Bas85] V. R. Basili. Quantitative Evaluation of Software Methodology. In *Proceedings of the First Pan Pacific Computer Conference*, Australia, July 1985.

[BBH93] L. C. Briand, V.R. Basili, and C. J. Hetmanski. Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Transaction on Software Engineering*, 19(11), November 1993.

[BCC92] V. R. Basili, G. Caldiera, and G. Cantone. A Reference Architecture for the Component Factory. *ACM Transactions on Software Engineering and Methodology*, 1(1), January 1992.

[BP84] V. R. Basili and B. T. Perricone. Software errors and complexity: An empirical investigation. *Communications of the ACM*, 27(1), January 1984.

[BP88] B. W. Boehm and P. N. Papaccio. Understanding and Controlling Software Costs. *IEEE Transactions on Software Engineering*, 14(10), October 1988.

[BR88] V. Basili and D. Rombach. The TAME Project: Towards Improvement–Oriented Software Environments. *IEEE Transactions on Software Engineering*, 14(6), June 1988.

[BR91] V. R. Basili and H. D. Rombach. Support for Comprehensive Reuse. *Software Engineering Journal*, 6(5), September 1991.

[Bro87] F. P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4), April 1987.

[BS87] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, 13(12), December 1987.

[BTH93] L. C. Briand, W. M. Thomas, and C. J. Hetmanski. Modeling and managing risk early in software development. In *Proceedings of the Fifteenth International Conference on Software Engineering*, May 1993.

[BWS93] J. Bailey, S. Waligora, and M. Stark. Impact of Ada in the flight dynamics division: Excitement and frustration. In *Proceedings of the 18th Annual Software Engineering Workshop.* NASA/GSFC, December 1993.

[CB91] G. Caldiera and V. R. Basili. Identifying and Qualifying Reusable Software Components. *IEEE Computer*, 24(2), February 1991.

[DK93] M. Dunn and J. Knight. Automating the detection of reusable parts in existing software. In *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, Maryland, May 1993.

[Dou87] D. Doubleday. ASAP: Ada Static Analyzer Program. Technical Report CS–TR–1897, University of Maryland, May 1987.

[EG90] N. Ebel and C. Genillard. The reusability of Ada software components. In R. Gautier and P. Wallis, editors, *Software Reuse with Ada*. Peter Peregrinus Ltd., 1990.

[End75] A. Endres. An analysis of errors and their causes in system programs. In *Proceedings of the International Conference on Software Engineering*, April 1975.

[GP87] A. Gargaro and T. Pappas. Reusability issues and Ada. *IEEE Software*, July 1987.

[Ich85] J. Ichbiah. *The Rationale for the Ada Programming Language*. Cambridge University Press, 1985.

[Jon84] T. C. Jones. Reusability in programming: A survey of the state of the art. *IEEE Transactions on Software Engineering*, SE–10(5), September 1984.

[Kes90] R. Kester. SEL Ada Reuse Analysis and Representations. In *Proceedings of the 15th Annual GSFC Software Engineering Workshop*. NASA/GSFC, November 1990.

[LG84] R. Lanergan and C. Grasso. Software Engineering with Reusable Designs and Code. *IEEE Transactions on Software Engineering*, SE–10(5), September 1984.

[LV89] R. Lind and K. Vairavan. An experimental investigation of software metrics and their relationship to software development effort. *IEEE Transactions on Software Engineering*, 15(5), May 1989.

[MK92] J. Munson and T. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5), May 1992.

[MP93] K. Möller and D. Paulish. An empirical investigation of software fault distribution. In *Proceedings of the First International Software Metrics Symposium*, Baltimore, Maryland, May 1993.

[Rom91] H. D. Rombach. Software Reuse: A Key to the Maintenance Problem. *Information and Software Technology*, 33(1), January/February 1991.

[Sel88] R. Selby. Empirically analyzing software reuse in a production environment. In W. Tracz, editor, *Software Reuse: Emerging Technology*. IEEE Computer Society Press, 1988.

[SEL94] An Overview of the Software Enginnering Laboratory. Technical Report SEL-94-005, Software Engineering Laboratory, NASA Goddard Space Flight Center, December 1994.

[SP88] R.W. Selby and A.A. Porter. Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis. *IEEE Transactions on Software Engineering*, 14(11), November 1988.

[Sta93] M. Stark. Impacts of object-oriented technologies: Seven years of SEL studies. In *Proceedings of Eigth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, October 1993.

[SYTP85] V. Shen, T. Yu, S. Thebaut, and L. Paulsen. Identifying error-prone software–an empirical study. *IEEE Transactions on Software Engineering*, SE-11(4), April 1985.

[TDB92] W. M. Thomas, A. Delis, and V. R. Basili. An evaluation of Ada source code reuse. In J. van Katwijk, editor, *Ada: Moving Towards 2000 (Proceedings of the Ada–Europe International Conference)*, Zandvoort, The Netherlands, June 1992. Springer-Verlag.

[WCW85] A. Wolf, L. Clarke, and J. Wileden. Ada-based support for programmming in the large. *IEEE Software*, March 1985.