# AN EVALUATION OF ADA SOURCE CODE REUSE*

W. M. Thomas, A. Delis & V. R. Basili

*Department of Computer Science*
*University of Maryland*
*College Park, MD 20742*

**Abstract**

This paper presents the results of a metric–based investigation into the nature and benefits of reuse in an Ada development environment. Four medium scale Ada projects developed in one organization over a three year period were analyzed. The study indicates benefits of reuse in terms of reduced error density and increased productivity. The Ada generic features are observed as an enabler of reuse at higher levels of abstraction. Finally, using several metrics, we identify trends indicating an improving reuse process.

## 1 Introduction

Reuse has long been cited as essential for obtaining significant improvement in software development productivity. Jones [16] indicates that only 15 percent of the developed software is unique to the applications for which it was developed. As development effort is often considered to be an exponential function of software size, a reduction in the amount of software to be created can provide a dramatic savings in development cost [8]. Reduced development cost is not the only benefit of reuse. Reused software has a track record—it has been well tested and exercised and thus may be more reliable and defect–free than newly developed software. The effect of the improved quality will not stop at the completion of development–rather the most significant benefit of reuse may be its effect on maintenance [18, 23].

To realize such benefits, techniques to achieve effective reuse have been the focus of extensive research effort over the past twenty years [24]. Generational approaches such as those described in [4, 5] attempt to achieve reuse through the generation of source code from other forms. Boyle and Muralidaharan [12] view the automatic translation as a successful mechanism to transfer programs into new programming environment. Repository based techniques strive for reuse by collecting reusable entities and providing efficient means to locate the appropriate object for a particular task. Techniques for storing objects to allow for effective automated retrieval are outlined in [2, 22]. Lanergan and Grasso [18] were able to provide for their organization a classification of functional modules in the context of the COBOL language and obtain a leverage of 60% of the regularly used code. Cheatham [25] outlines a methodology of abstract programs that can be *instantiated* to a family of concrete programs using very high level languages. Some other attempts geared predominantly towards source code reuse are found in [19, 20]. The common element in

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

these efforts is that they all strive for the reuse of products or by-products of the software life cycle. Basili et al. [6, 7] indicate that the reuse of processes in addition to software products may result in even greater benefits.

Caldiera and Basili [13] identify four fundamental steps in a reuse process cycle and introduce the idea of metric use for the identification and extraction of reusable code. A validation study was performed focusing on the identification of reusable components in a C/Unix environment.

In this paper we discuss the use of measurement to better understand and evaluate an Ada reuse process. Using various metrics, we analyzed the effect of reuse in Ada developments in a single organization over a 3 year period. This paper extends the work presented in [13] in the Ada environment and generalizes the approach of using software metrics to determine the effectiveness of reuse. We argue that metrics, in addition to facilitating the extraction of software components, can aid in the evaluation of reused code and reuse processes. We have performed an investigation into the nature and relative benefits of reuse in the Software Engineering Laboratory (SEL). The SEL is a joint effort of the NASA/Goddard Space Flight Center, the University of Maryland, and the Computer Sciences Corporation to study software engineering issues and promote modern development techniques.

The paper is organized as follows. Section 2 describes the use of metrics for the assessment of a reuse process. Section 3 describes the environment that was analyzed. Section 4 presents the results of the analysis in three areas: the resulting effect of reuse in the development environment, how well the Ada generic constructs support reuse, and what trends can be seen as the organization gains reuse experience. Section 5 summarizes and identifies major conclusions.

## 2 A Measurement Guided Reuse Process

Basili and Rombach [7] outline a framework for the support of a reuse oriented development environment. The framework consists of a reuse model, describing how objects are taken from a repository to their new context, a characterization schemes for the model, allowing for effective use of the model, and an environment model supporting the integration of reuse into the development environment.

A project organization can be tailored toward reuse by separating project concerns from software component development concerns. Such an organization is described in detail in [13]. The distinction with traditional development is that in this reuse oriented organization, the project organization provides specifications to the factory organization, which retrieves the appropriate components, and provides them to the project organization for integration. As shown in figure 1, the project side of the development is relieved of the of the work in developing components, rather, it must specify, design, and integrate components into a working system. The item of concern in this organization is the system, not the components. Component release takes place in the factory side of the organization. Components requested by the project organization can be created new, or reused from a component repository. Thus activities involved in the factory side include searching for components, adapting and creating components, qualifying and storing the components. This side is focused on the component rather than the system into which the component must be integrated. It is not expected that the factory side is driven solely by project requests; rather domain analyses will sustain continuing development in the factory.

The work of Caldiera and Basili [13] deals with the factory side of the organization. The major goal of that work is the ability to locate potentially reusable components in existing software using

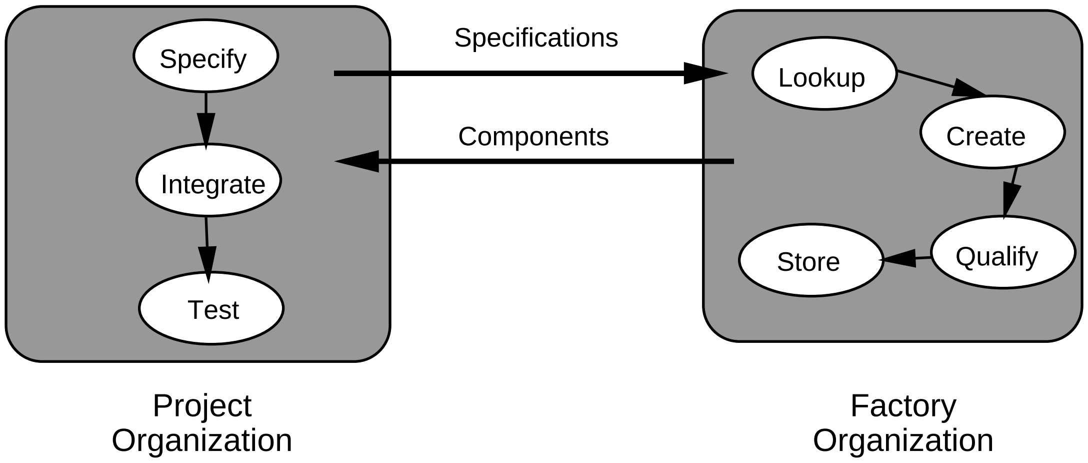

Figure 1: A Reuse-Oriented Software Development Organization

tuples of software metrics for the qualification of the candidates. Once identified as a candidate, the component is re-engineered in to an object suitable for the repository. This re-engineering may be in making the component more general, removing certain domain dependencies, or in adding a detailed specification.

Ada provides support for reuse in several ways. Separation of specifications from bodies allows the developer to separate the concerns of the interface from the implementation. Packages provide for the encapsulation of related entities, and facilitate the creation of abstract data types. Generic program units provide parameterized program templates that enable the adaptation of an abstraction to a variety of contexts.

Even with such language features, the transition to a reuse–oriented organization will be a gradual one. As more and more experience is captured and packaged, we expect a decrease in effort spent in the product life cycle, an increase in effort spent in the factory cycle. Throughout the transition the effectiveness of reuse can be monitored. Measurement can help us to better understand, evaluate and improve the quality of the reuse processes. A general architecture supporting the integration of measurement with software development processes is outlined in [7]. The reuse-oriented organization can only be effective if it shows an overall improvement over a traditional organization. By monitoring the resulting effect on the cost and quality of the final product, we can better understand the reuse process and evaluate its effectiveness.

Our goal is to examine, through the use of various metrics, the development side of the cycle to understand how reuse facilitates Ada development. In particular, we wanted to learn what benefits can be achieved relative to reuse in terms of quality and cost. A second goal was to examine the use of the Ada generic features to determine how well they support the reuse process. Finally, we try to identify trends that can be observed as the organization gains reuse experience. To determine the effectiveness of reuse in the environment, we examined error density, program complexity, and productivity relative to reuse rates. We examined the use of generics relative to error density and complexity to better determine how the Ada language features were being used in the context of reuse.

## 3 Description of the Experiment

We analyzed a collection of four medium-scale Ada projects developed at the NASA/Goddard Space Flight Center. The projects ranged in size from 35 to 75 thousand non-comment non-blank source lines of code (KSLOC), and required development effort of 30 to 175 technical staff months. We analyzed reuse from two perspectives, the first, off-the-shelf code reuse of previously developed compilation units, and the second, reuse of functionality achieved with generic instantiations. Another direction that was taken was to investigate the project development over time, and assess how experience acquired to date has contributed to greater reuse achievements in the SEL environment. The percentage of reused code on these projects ranged from 9 to 87 percent (verbatim), and 29 to 94 percent (verbatim and with modification) [17].

The purpose of this experiment was to quantify reuse with objective metrics, assess the impact of reuse on complexity, and to investigate the viability of a metrics based approach to reuse in an Ada development environment. The NASA/GSFC SEL has collected a wealth of data on software development over the past fifteen years. We used several types of data in our analyses. The first type of data has to do with the origin of a component—whether it was newly created or reused. For each component in the system a component origination form is filled out by the developer, identifying the origin as one of four classes: newly created, reused with extensive modification (greater than 25% of the SLOC modified), reused with slight modification (less than 25% of SLOC modified), and reused verbatim. For Ada systems, a compilation unit is viewed as a component. Thus our analysis focuses on looking at the different classes of compilation units.

Our second notion of reuse is the reuse achieved via the instantiation of generic units. For this analysis, we split our system into two parts, the generic part, consisting of units associated with generic library units and instantiated library units, and the non-generic part, consisting of units associated with all other library units. As with custom versus reused code, we believed we would see differences between the two classes.

We analyzed the systems with a source code static analysis tool, ASAP [14], 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. ASAP also identifies all **WITH** statements, so we were able to develop a measure of the externals visible to each unit.

Two measures of development productivity were analyzed, namely, the productivity associated with the code/unit test phase, and the productivity in the system and acceptance test phases. We also use, as a measure of quality, counts of development error reports for each compilation unit.

## 4 Results

In this section we discuss results of our analysis in three areas. The first compares reused with newly created code from the perspective of the resulting effect on product quality and development effort. We then analyze how well the Ada Generic features facilitate reuse. Finally, we discuss how measurement can be used to identify trends in reuse over several projects.

### 4.1 Effect of Code Reuse on Product Development

Reuse has been advocated as a means for reducing development cost and improving reliability.

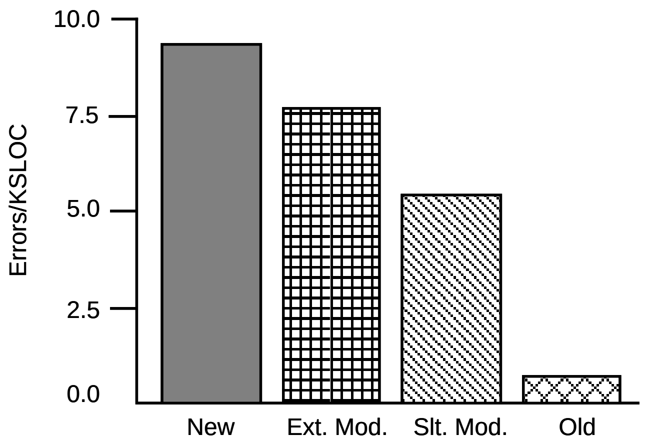

Figure 2: Error Density by Class of Reuse

Boehm and Papaccio [9] identify the reuse of software components as one of the most attractive strategies for improving productivity.

The following sections describe what we have seen relative to reuse across four recent projects in three areas: error density, product complexity, and productivity.

### 4.1.1 Error Density

Rework has been identified as a major cost factor in software development. Jones [15] indicates that it typically accounts for over 50% of the effort for large projects. Reuse of previously developed, tested, and qualified components can reduce the number of errors in development, thus reducing rework effort.

Figure 2 shows the error density found in each of our four reuse classes over the four projects analyzed. The bar labeled "New" indicates error density in newly created components. "Ext. Mod.", "Slt. Mod." and "Old" refer to error densities found in the components in the classes of Extensively Modified, Slightly Modified, Reused Verbatim, respectively. As expected, fewer errors were found associated with reused code vs. new code. Error density (Errors/KSLOC) was found to be 0.9 in the code reused verbatim, 5.6 in the slightly modified code, 8.1 in the extensively modified code, and 9.5 in the newly developed code. The lower density in the reused components implies an easier time in integrating the reused components into a new system than in developing code from scratch. However, it also appears that there is a significant difference in error densities of the modified code compared to the verbatim code. In fact, there seems to be no difference in error density between components that were developed new and those reused with extensive modification. The slightly modified code shows a 40% improvement in error density relative to newly developed code; however the most significant benefit comes with the unchanged components, as they show a 90% reduction relative to the new components. Clearly, the greatest benefit comes from reusing the code without modification.

### 4.1.2 Product Complexity

A software system can be viewed as an inter-related collection of components. The quality of the system thus is a function of the quality of both the components and the component relations.

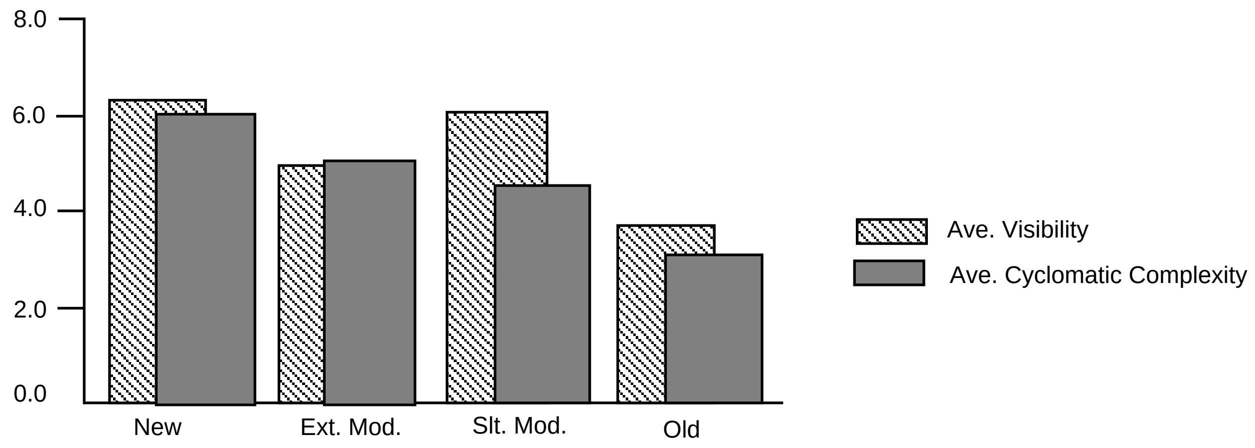

Figure 3: Relationship of Reuse and Program Complexity

Cyclomatic complexity [21] is a graph–theoretic measure of the control organization of a component. Highly complex components may be more error-prone and difficult to understand. In the context of reuse, excess complexity is seen as an inhibitor of effective reuse, as it can make the integration and rework costs outweigh the cost of developing a component from scratch [13]. The complexity of the component interface is also an important factor. Limiting program dependencies is suggested as a means for improving reusability, and techniques for transforming existing software to limit dependencies is discussed in [3]. Agresti et al. [1] have developed multivariate models of software qualities using characteristics of the software architecture. Increasing dependencies in the system is shown to reduce the reliability of the system. There is a trade–off between these two complexities. One can achieve a simple interface complexity at the expense of increased internal complexity, and vice versa. Developers strive for the proper balance of these complexities.

To assess the internal complexity of the components in the systems, we examined the cyclomatic complexity of the executable program units. As a rough measure of the interface complexity of the compilation unit we observed the number of library units that are made visible to the compilation unit. Figure 3 shows the relationship between reuse, cyclomatic complexity, and visibility. Among the executable program units, the mean program unit cyclomatic complexity was lower for reused components than for the new components. New components had an average complexity of 6.4, extensively modified 5.1, slightly modified 4.4, and unchanged 3.6. A nonparametric test of the significance of the difference in the class medians indicates the difference to be statistically significant at for all pairs of classes except in distinguishing between new and extensively modified components. The overall relation ship of reuse with project complexity is not so clear. While there seems to be a significant drop in average complexity (over the entire system) from the first project to the second, there is only a slight decrease in each of the subsequent projects.

In terms of the visibility to the compilation units, we observed a lower average number of visible library units in the verbatim reused components. This average visibility was found to be 6.3 in the class of new components, 5.0 in the extensively modified, 6.0 in the slightly modified, and 3.6 in the reused verbatim. While there was a significant difference (at .0001 level) between the visibility in the reused verbatim class and each of the other classes, no such distinction was found when comparing the classes of new, slightly modified and extensively modified components. This supports the view presented in [3] that reducing dependencies may make a module more reusable.

One possible explanation for the lower complexity observed in the reused code is that the reused components comprise only simple, straightforward functions (e.g. general utilities), and as

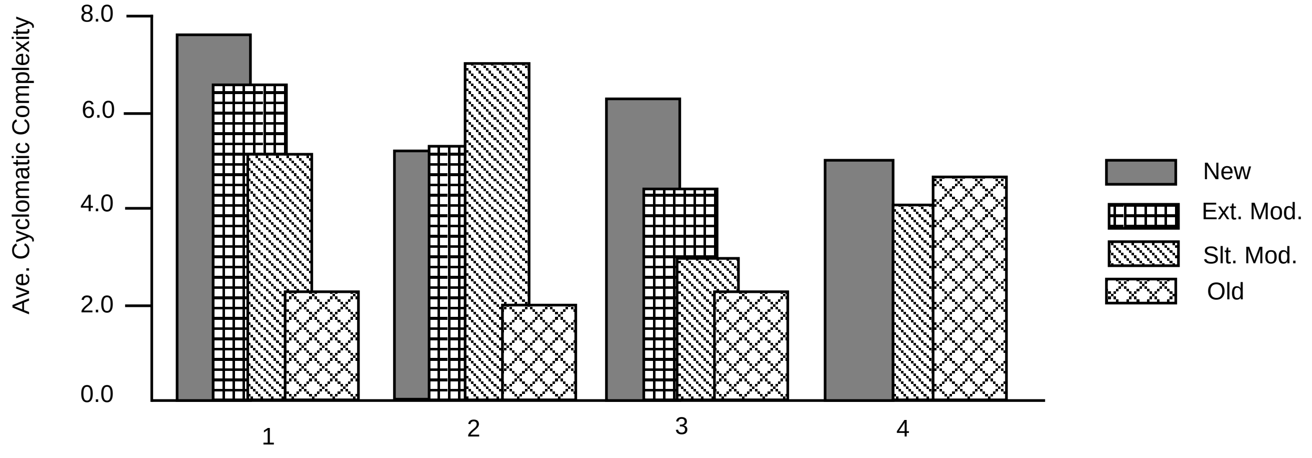

Figure 4: Complexity of Reused Objects Over Time

such, should have lower complexity and fewer dependencies. However, in this environment we saw an increasing level of reuse from project to project, and a decreasing complexity over the entire system. Figure 4 shows the relationship between reuse and complexity from the four observed projects. We see reuse of increasingly complex objects, both in terms of their internals and their interfaces, but at the same time see an overall reduction in the complexity of the entire system. While it certainly is true that general utilities are being reused, it also is evident that there is a trend toward the reuse of more complex functionality.

### 4.1.3 Productivity

We also analyzed productivity across the projects from the perspective of reuse. In particular, we wanted to see if reuse would provide a significant reduction in effort in the testing phases, as well as in the coding phase. We defined implementation productivity as thousands of non–comment, non–blank source lines of code divided by staff-months (KSLOC/SM) charged during the code/unit test phase, and test productivity as KSLOC divided by staff-months charged during the system and acceptance test phases.

Figure 5 shows that as reuse increases, productivity increases both in the implementation phase and the test phase. The lines indicate the (log of the) percentage of reuse, both verbatim reuse and total reuse, and the bars indicate implementation and test productivities. Implementation productivity ranged from 2.5 KSLOC/SM on the project with the least reuse (26%) to 5.8 KSLOC/SM for the project with the most reuse (94%). It was expected that reuse would have a significant positive impact on effort expended in the implementation phase. We observed a similar result with respect to productivity in the system and acceptance test phases, as we saw a productivity range from 2.5 to 6.5 KSLOC/SM. While this data is not sufficient to build an accurate model relating reuse and cost, it does provide an indication that the effect of reuse is widespread—in addition to the savings in the implementation phase we see an indication of savings in integration and test phase.

## 4.2 Generics vs. Non–Generics

Booch [10] identifies the primary use of generic units as reusable software components. The parameterization of generic units allows them to be applied to a range of uses, and thus can reduce

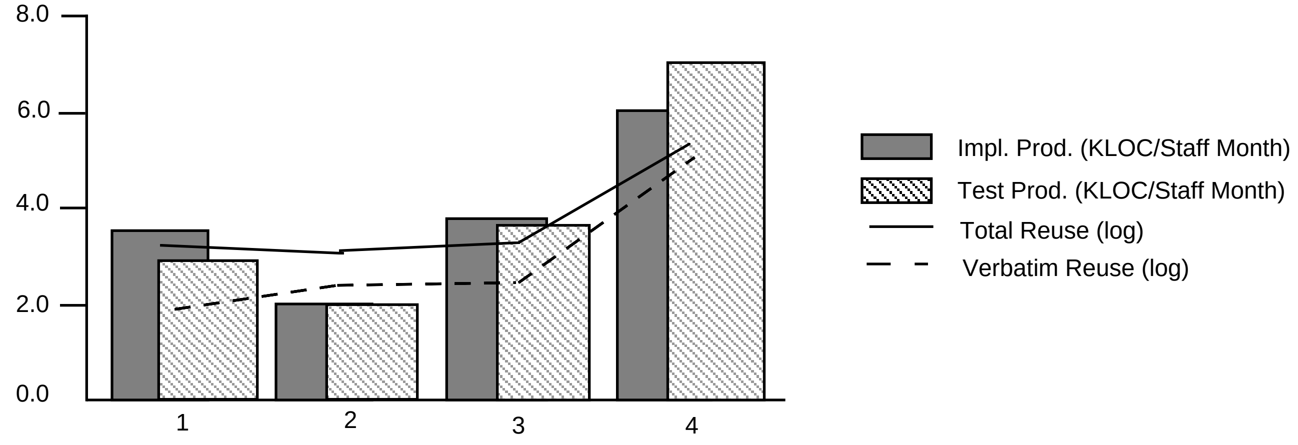

Figure 5: Effect of Reuse on Productivity

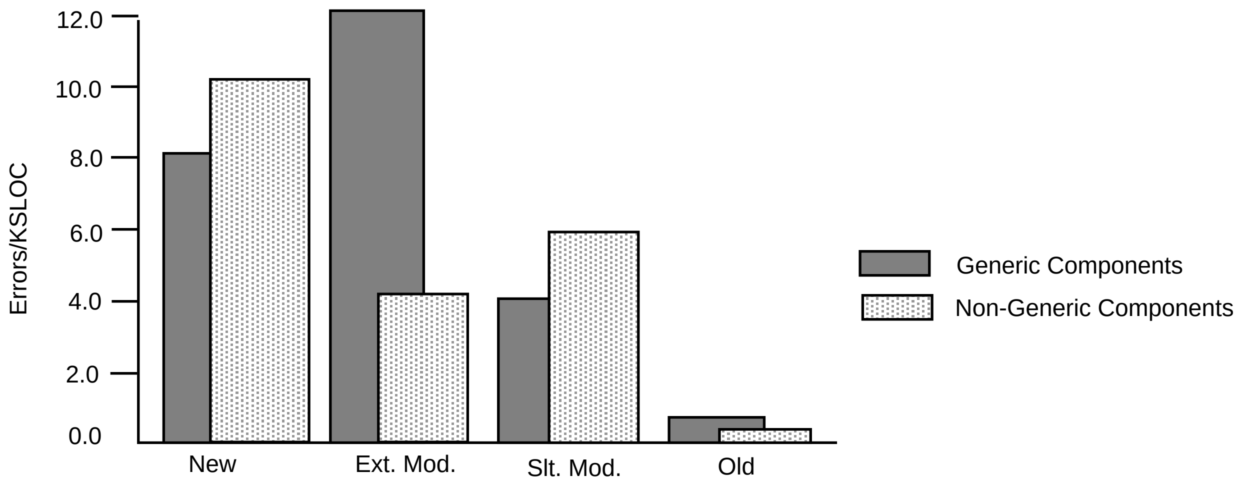

Figure 6: Error Density Profile in Generic and Non–Generic Components

the amount of code that needs to be written. One goal of this experiment was to see whether the increased effort in making a component generic would manifest itself with higher error densities or increased complexity. To investigate this, we divided the components in two classes - generics, consisting those compilation units related to generic library units either by instantiation or as a secondary unit, and non-generics, consisting of the others. Ignoring the origin of the components, we see a lower error density in the generics, relative to the non-generics, as the error density was 4.6 errors/KSLOC in the generic part, and 7.7 in the non generic part. We expected a significantly lower error density in the generic part, since it may contain a greater proportion of reused software. When we further distinguish between classes of component origins, as described previously, we see more interesting patterns. Figure 6 shows the error density profile in each class of component origin for both the generic units and the non-generic units. Overall, among the newly developed units, we found a significantly lower error density in the generic components (8.0 errors/KSLOC) than in the non-generic components (10.1). Among the components reused with modification, the overall error density is 6.6. For the generics, we found a slightly higher error density, 7.5 errors/KSLOC, vs. 5.7 for the non-generic part. Among the reused verbatim components, overall error density is very low (0.7 errors/KSLOC), and there was little difference in the error densities associated with the reused generics versus reused non-generics.

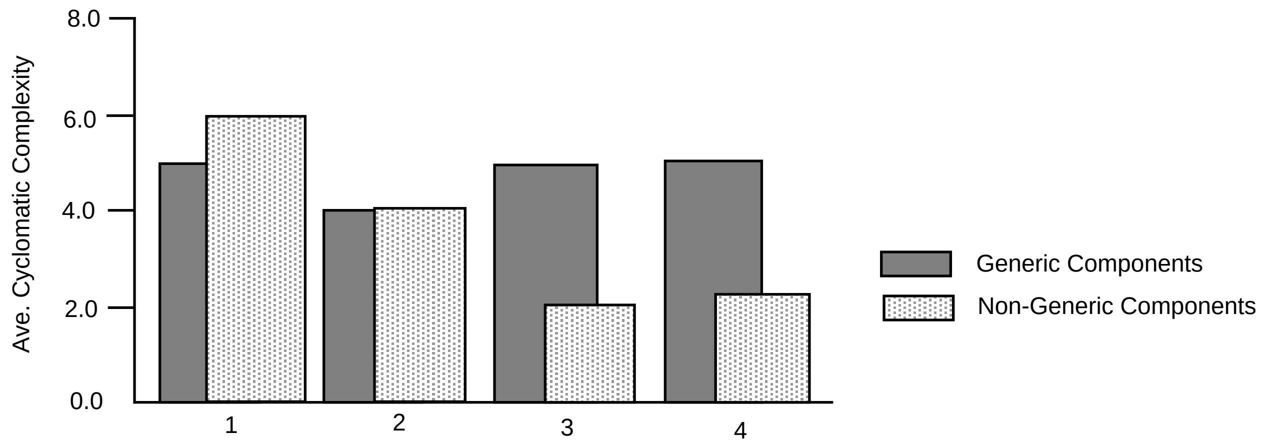

Figure 7: Cyclomatic Complexity in Generic and Non-Generic Components

These results may be interpreted as follows: either that the developers of the generic units took more care in their creation, and thus made fewer mistakes, or that the generic components were simple units to develop, and should have fewer errors associated with them. Further analysis of the complexity of the newly created generics over time show that more and more complex objects are being created generically, and simpler objects being created as non-generics. Figure 7 depicts this trend. As the generic proportion of the software increases, we see little change in the complexity of the generic part, while observing a significant reduction in the complexity in the non-generic part.

There is a different pattern in the generic components when comparing error density by origin. Among the modified components, there is a lower error density than in the new components. However, in the class of modified components we see a higher error density in the generics than in non-generics. This suggests a greater difficulty in modifying generic components than in non–generic.

## 4.3 Reuse Trends

Our final purpose was to examine the effect of reuse over time using a metric–based approach. The change to a more generic architecture is evident from the percentage of the generic portion of the system. This change provided significant benefits—a reduction in development cost, duration, and error density [11]. Looking across all projects, we see few differences among the classes in terms of the complexity metrics. However, when we look at each project individually, we see a changing pattern of reuse. Figures 8 and 9 show the changing profile of reuse over time. The class of custom components includes both the newly created and extensively modified components, while the reused class includes slightly modified and verbatim reused components.

Across the four projects we see a slight trend of increasing complexity in the reused components, and decreasing complexity in the custom components, both in terms of internal (cyclomatic) and external (visibility) complexities. This suggests that as an organization packages more and more domain experience, the complex objects will be reused, and relatively simple objects will be newly created to join them together. This is supported by the falling complexity in the custom, non-generic components (i.e. the most application specific components). We do not see such a clear pattern in the visibility of the custom non generic components. This may simply be evidence that these components are still being created at a relatively high level in the application hierarchy. When we examine the complexities of the reused components, we see a trend of increasing complexity

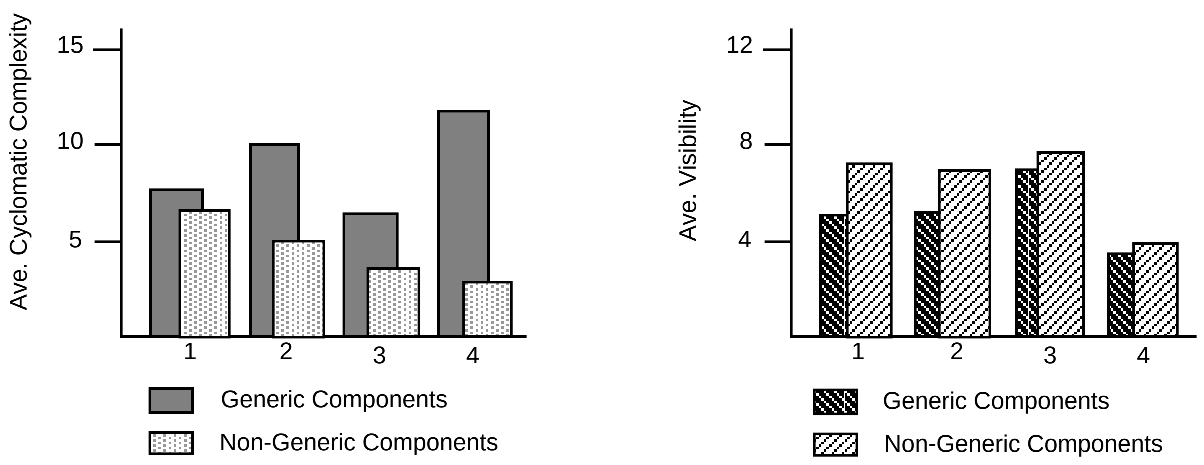

Figure 8: Complexity Trends in Custom Components

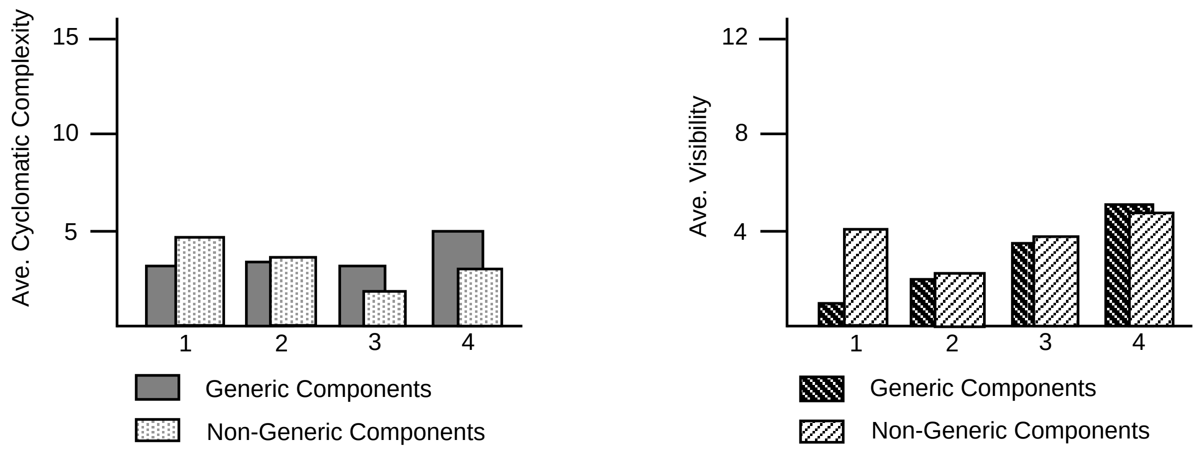

Figure 9: Complexity Trends in Reused Components

in the reused generics, both in terms of cyclomatic complexity and visibility. This may indicate that more complex objects (including those at higher levels in the application hierarchy) are being reused.

Analysis of the complexities of the objects over time illustrates the improvement of the reuse process. Object-Oriented design and the Ada language features may be a primary reason for the improving reuse process in this organization. The increased complexity of the reused objects suggests that reuse is occurring at higher levels of abstraction. This supports the notion that a model of reusability must be evolved over time to keep pace with a changing environment [13]. While the lower cyclomatic complexity and visibility associated with the reused components indicates that such measures may work well in the initial assessment of reuse candidates. Clearly, however, reuse with Ada is not limited to these types of candidates.

## 5 Conclusions

In this paper we present a metrics-based process to assess Ada reuse. We analyzed an Ada development environment from a reuse perspective and found significant benefits of increasing reuse. Productivity increased and error density was reduced. As expected, we observed that while reduced error densities (compared to newly created components) can be achieved both in verbatim reuse and in reuse with modification, a much more substantial reduction occurred with the verbatim reuse. This supports the view that the most benefit from reuse comes from direct reuse without modification.

The Ada package and generic constructs enable effective reuse within an application domain. The marked improvement in verbatim reuse has shown substantially lower error rates and development effort. The adoption of a generic architecture in the SEL [11] clearly has resulted in an improved, reuse-oriented development. We see no indication that generics are significantly more difficult to develop than non-generics, in fact we have seen lower error densities in newly developed generic components than in newly developed non-generic components. In terms of reuse of the generics, we observed little difference between error rates associated with verbatim reuse of generics and non-generics. However, when looking at the modified components, extensive modification of the generics was seen to be significantly more error prone than extensive modification of non-generic components.

Finally, we have indications that metrics can be used to show trends of an improving reuse process. The increased use of generics has resulted in the creation of simpler custom components, and allowed the reuse of more complex components.

## 6 Acknowledgment

We would like to thank Frank McGarry of NASA/GSFC and William Agresti of the Mitre Corporation for their help in the realization of this study.

## References

[1] W. W. Agresti, W. M. Evanco, and M. C. Smith. Early Experiences Building a Software Quality Prediction Model. In *Proceedings of the Fifteenth Annual Software Engineering Workshop*, NASA/GSFC, Greenbelt, Maryland, November 1990.

[2] N. Badaro and Th. Moineau. ROSE–Ada: A Method and a Tool to Help Reuse of Ada Codes. In *Ada: The Choice for '92 (Proceedings of the Ada–Europe International Conference)*, Athens, Greece, May 1991.

[3] J. Bailey and V. Basili. Software Reclamation: Improving Post-Development Reusability. In *Proceedings of the Eighth National Conference on Ada Technology*, 1990.

[4] R. Balzer, T. Cheatham, and C. Green. Software Technology in the 1990's: Using a New Paradigm. *IEEE Computer*, 16(11), November 1983.

[5] D. Barstow. Rapid Prototyping, Automatic Programming, and Experimental Sciences. *Software Engineering Notes*, 7(5), December 1982.

[6] V. Basili, D. Rombach, J. Bailey, and A. Delis. Ada Reusability Analysis and Measurement. In *Proceedings of the 6th Symposium on Empirical Foundations of Information and Software Sciences*, Atlanta, Georgia, October 1988.

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

[8] B. W. Boehm. Software Engineering Economics. *IEEE Transactions on Software Engineering*, SE–10(1), January 1984.

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

[10] G. Booch. *Software Engineering with Ada.* Benjamin–Cummings, second edition, 1987.

[11] E. W. Booth and M.E. Stark. Designing Configurable Software: COMPASS Implementation Concepts. In *Proceedings of Tri-Ada 1991*, October 1991.

[12] J. Boyle and M. Muralidaran. Program Reusability through Program Transformation. *IEEE Transactions on Software Engineering*, SE–10(5), September 1984.

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

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

[15] C. Jones. *Programming Productivity.* McGraw–Hill, 1986.

[16] T.C. Jones. Reusability in Programming: A Survey of the State of the Art. *IEEE Transactions on Software Engineering*, SE–10(5), September 1984.

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

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

[19] S. Litvintchouk and A. Matsumoto. Design of Ada Systems Yielding Reusable Components: An Approach Using Structured Algebraic Specification. *IEEE Transactions on Software Engineering*, SE–10(5), September 1984.

[20] Y. Matsumoto. Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels. *IEEE Transactions on Software Engineering*, SE–10(5), September 1984.

[21] T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4), December 1976.

[22] R. Prieto-Diaz and P. Freeman. Classyfying Software for Reusability. *IEEE Software*, 4(1), January 1987.

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

[24] T. Standish. An Essay on Software Reuse. *IEEE Transactions on Software Engineering*, 10(5), September 1984.

[25] Jr. T. Cheatham. Reusability Through Program Transformations. *IEEE Transactions on Software Engineering*, SE–10(5), September 1984.