Error Density and Size in Ada Software

Carol Withrow, *Unisys Communication Systems*

The error density in program modules appears to be lowest at an optimum intermediate size, as this empirical study shows. For Ada, that size is about 225 lines.

Growing awareness that software engineering needs to improve its products and incorporate more rigor into its processes has recently led to an increased interest in metrics. It has long been recognized that the ability to measure a process or product increases the understanding of it. As Lord William Kelvin, the British mathematician and physicist, said, "When you can measure what you are speaking about, and express it in numbers, you know something about it." On the other hand, his contemporary, the biologist Thomas H. Huxley, warned, "As the grandest mill in the world will not extract wheat flour from peascods, so pages of formulas will not get a definite result out of loose data."[1] This was a 19th-century formulation of "garbage in, garbage out."

With this caveat in mind, consider the value of two measures: module size and error density.

Size is the oldest measure of software complexity, which is believed to be a major driver of the maintenance effort. While more sophisticated complexity measures have been devised, no universal tool has emerged, and size — defined here as source lines of code — continues to be popular.[2] Although there are many problems with this metric, such as the definition of a source line and the difficulty of comparisons across languages or across programming styles, its popularity is justified on the bases of ease of determination and a track record of success in effort estimation.[3,4]

Error density is popularly believed to be directly related to program size. James Martin has said that there is a disproportionate increase in complexity with increasing size, implying a similarly disproportionate increase in errors.[5] Unfortunately, I could find no published study to support this view, although T.A. Thayer and colleagues[6] have hinted at the possibility. Nevertheless, this direct relation-

0740-7459/90/0100/0026/$01.00 © 1990 IEEE

ship seems to be what you would intuitively expect. As a module increases in size, it becomes more difficult for the programmer to manage the increasing number of details, which leads to the generation of errors.

However, a higher error density occurred in smaller modules in studies by Vincent Shen and colleagues [7] and by Victor Basili and Barry Perricone.[8] Shen and colleagues found that an inverse relationship existed up to about 500 lines. Beyond that they found no relationship, but they had a small sample of modules larger than 500 lines. They studied software written in Pascal, PL/S, and assembly language. Basili and Perricone examined Fortran modules sized mostly at fewer than 200 lines. They attributed the inverse relationship primarily to the predominance of errors at the interfaces between modules, such as incompatible usage of common data. These interface errors were spread equally across all modules regardless of size and thus represented a higher density in smaller modules. Other possible causes they noted were that most modules examined were small (causing bias), that extra care may have been taken in coding larger modules, and that there may have been undetected errors in larger modules.

My colleagues at Unisys and I had an opportunity to contribute to this area, having just completed the development of a large project written in Ada. The project team kept error reports, and the many modules ranged greatly in size. We analyzed module size to see if there was a relationship with module quality. We used error density — defects per thousand lines — as an inverse measure of quality: the lower the error density, the higher the quality.

Method

The object of our study was the Ada software for the command and control of a military communications system. It was developed by a team of about 17 experienced programmers to whom Ada was a new language.

We computed the error density for each of the 362 modules, which totaled 114,000 lines. (Lines are defined here as noncomment lines. There were more of these in the Ada software studied than there were Ada statements, since a statement often spanned more than one line. To make these results more comparable to published work on error density, we converted the statement count to lines by making a manual count of a representative sample of modules to determine the ratio of actual lines to Ada statements. We then applied this factor, 1.77, to the statement counts determined by an automated count of semicolons.)

The direct relationship between number of errors and program size seems intuitive. But other studies also showed that small programs are error-prone. Why?

The project chose to treat Ada packages — rather than procedures, functions, subprograms, or tasks — as modules because the package is the basic structuring unit of the Ada language and because this software was developed on a package basis.

The data are based on problem/change reports written during the test and integration phases — beginning immediately after code and unit testing and ending with the delivery of the product to the customer. Error records for the period before this were not kept and do not yet exist for the period after it.

The problem/change reports, originally numbering 696, were reduced to 491 by eliminating those that were not true software errors. Those eliminated included changes required by changed requirements, document changes, and errors caused by hardware. The problem/change report count was then further reduced by four to 487 by eliminating those with incomplete information.

An error here is defined as a package being changed because of a problem/change report, so if four report errors were made for a package, the error count for that package is four, although the number of lines of code changed may well be greater than four. The number of errors found per package ranged from zero to 39. The average was two errors per package. The average package size was 316 lines, with a range of four to 5,160 lines. Of the 362 modules in the study, 137 (38 percent) had at least one error.

Results

Figure 1 shows a scatter diagram of the resulting error densities plotted with size as the independent variable. The data are from the packages with nonzero error densities only. This diagram bears a striking resemblance to one published by Shen and colleagues. There is a regular pattern of declining error density with increasing size, but the regularity appears to be partly an artifact of the discrete nature of the data. The empty area in the lower left represents the impossible case of fractional errors. The lower limit of one error per module manifests itself as an orderly, curving row of data points. To highlight this, Figure 2 shows the same data converted to log (base 10) values. The curve for one-error modules is here a straight row; you can discern similar rows for two-error modules and so on.

However, to characterize the data com-

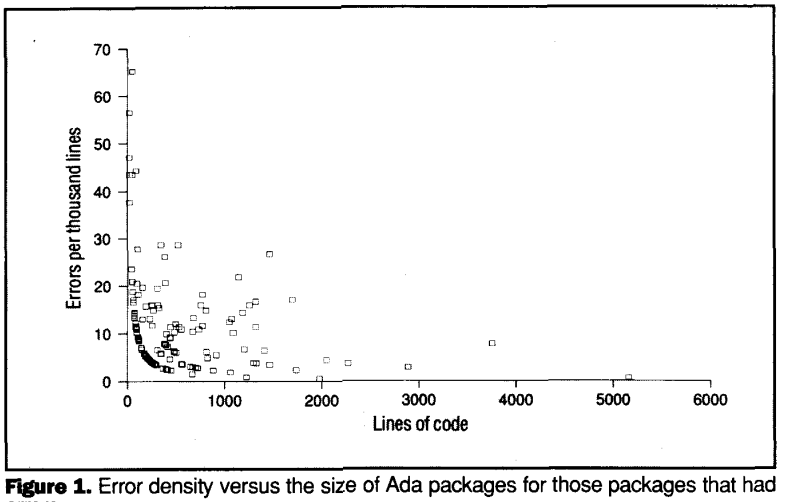

Figure 1. Error density versus the size of Ada packages for those packages that had errors.

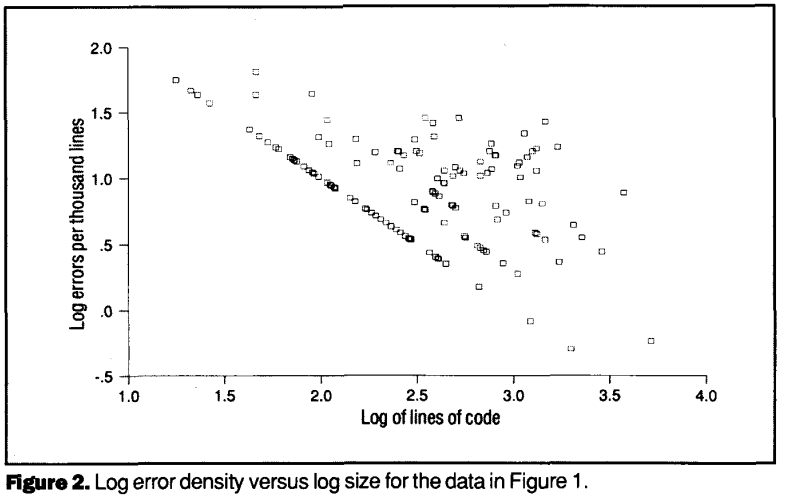

Figure 2. Log error density versus log size for the data in Figure 1.

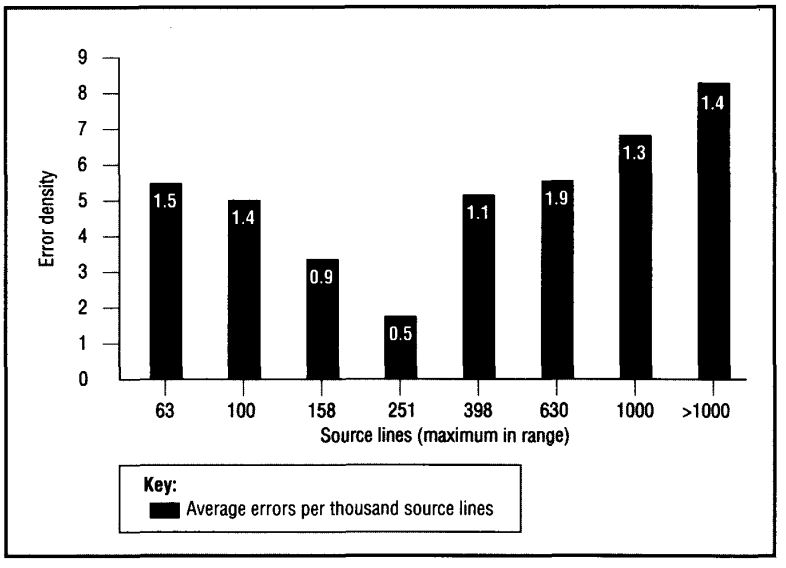

Figure 3. Error density versus size for all packages (those in Figure 1 plus those with no errors) in bar-chart format. The standard error of the mean represented by each bar is shown at the top of the bar.

pletely, the error-free modules must be included. This of course produces many points along the x axis. You can perform no significant curve fitting on these data, so we used a bar graph to show the average error density found in all modules grouped by size (Figure 3). We chose the size categories by equal log spacing of the number of lines. The numbers on the x axis represent the largest number of lines in the collection of modules represented by the bar above. For example, the bar labeled "100" represents modules with 64 to 100 lines, and the bar labeled "158" represents 101 to 158 lines. The values on top of the bars show the standard errors of the values represented by the bars.

This graph shows the pattern postulated above: falling error density for small modules followed by rising error density for larger modules, with an apparent optimum size of 200 to 250 lines. Table 1 shows detailed statistics for the data of Figure 3.

Comparing studies

This result lends support to the hypothesis that there is an optimal, intermediate module size. This might be dubbed the Goldilocks principle after the children's fable and suggests that software designers may decrease how error-prone their products are by decomposing problems in a way that leads to software modules that are neither too large nor too small.

Differences. Basili and Perricone's study showed a declining error rate, with the lowest rate occurring in the largest category (code larger than 200 lines). The main differences between their results and our results are in the sample size of large modules and the method of counting lines. Basili and Perricone had only three modules larger than 200 lines, while half the modules in our study were that large. Also, Basili and Perricone counted only executable lines (in Fortran) and not common data statements, while our study counted both declaration and body lines of source code (in Ada). (Neither study counted comment lines.) Therefore, their line counts may have underestimated size when compared to our study. Another difference is the part of the life cycle studied: Basili and Perricone covered the coding through the mainte-

Table 1.
Error-density statistics for packages grouped by size.
Groups were chosen for equal log spacing of source lines.

Package maximum source lines	Log_{10} maximum source lines	Range of source lines	Number of packages	Mean error density	Standard error
63	1.8	4-62	93	5.4	1.5
100	2.0	64-97	39	4.9	1.4
158	2.2	103-154	52	3.4	0.9
251	2.4	161-250	53	1.8	0.5
398	2.6	251-397	46	5.2	1.1
630	2.8	402-625	31	5.6	1.9
1,000	3.0	651-949	22	6.8	1.3
>1,000	>3.0	1,050-5,160	26	8.3	1.4

nance phases, while our study covered only the integration and test phases.

Shen and colleagues, who also found a declining error rate with size, studied a compiler written in PL/S, a derivative of PL/I. They showed this inverse relationship to hold up to a size of about 500 lines. They did not specify their definition of lines. The life-cycle phases they studied were integration through maintenance.

Thayer and colleagues studied a large command and control system written in Jovial J4, an Algol derivative, and found no correlation between module size and the density of the errors detected during preoperational testing. Preoperational errors constituted the bulk of errors studied. However, limited data on operational errors suggested a higher error density in routines of 1,000 to 2,500 lines.

But, given all the differences in phase, language, application, and measuring technique, the surprising outcome is the concordance of results among studies.

Explaining the curve. You can view the shape of the graph of error density versus size of all modules as an overlay of two phenomena.

The reason for the rising tail is intuitive: Complexity increases with module size, inviting errors. There is a limit to the amount of code a programmer can hold in his immediate frame of attention. When the module size exceeds this, errors of oversight are more likely. Thus, the rising tail of the curve is understandable.

However, the initial falling portion of the curve seems to be counterintuitive. The scatter graph of error density versus size for modules that had at least one error shows that an artifact of discrete data contributes to the impression of falling error density. Only by including the error-free modules do you get a true picture, but even then the phenomenon of falling error density persists.

Two possible reasons are related to testing. When just one part of a module is tested, other parts are to some degree also tested because of the connectivity of all parts. Thus, in a larger module, some parts may get some free testing as testing proceeds through the early phases that precede error reporting. Or perhaps the larger modules contain undiscovered errors because of inadequate testing of all paths. Another possible cause of falling error density is that interface errors tend to be evenly distributed among modules and, when converted to error density, they produce larger values for small modules.

An interesting analogy is a study by Rajiv Banker and Chris Kemerer, who examined software quality and size on a different scale, based on software projects as a whole rather than on software modules.[9] They examined the economy or productivity of many software projects and related this to project size. The result was that some investigators reported increasing economy with increasing size, while others reported decreasing economy with increasing size.

Banker and Kemerer reconciled these conflicting results with a hypothesis that said for any environment there is an optimal product size. For lesser sizes there is rising economy, and for greater sizes there is declining economy. They attributed the rising economy primarily to the spreading of fixed overhead; they attributed the declining economy to proliferating communication paths.

There appears to be a similarity between this phenomenon and the module-based data reported earlier. The fixed overhead of the project is comparable to the relatively constant factor of interface errors in the modules. While the overhead is spread over the components of the project, the interface errors are spread over the lines of code when converted to error density. On the declining side of the economy curve, the proliferating communication paths (human) of the project are clearly analogous to the proliferating communication paths (computational) of the module.

We thus propose a synthesis of the two views of module error density into one larger view that is analogous to the view Banker and Kemerer put forth for project-based data. This new view is one of falling error density, followed by rising error density for increasing module sizes. This provides a framework for viewing the surprising results of Shen and colleagues and of Basili and Perricone as credible phenomena.

Ada peculiarities. Comparing this study with others, no result that might be uniquely attributable to the Ada language is evident. Given the many error-reducing features of Ada — data abstraction, strong typing, information hiding, generics, and dynamic binding — you might expect that error characteristics, including the pattern of error density versus size, would be different for this software. The Ada compiler detects certain classes of errors that can persist through testing in other languages. If the high error density of small modules is due to interface errors, this should be lower for Ada software because the compiler checks for some of this.

However, although the types of errors may be different, their overall distribution by module size does *not* conflict with what little is known about error density in conventional software. But the newness of Ada to the development team may have skewed the results in some way not yet understood.

Our results and those of others suggest a new view of error density: that it declines as module size increases up to some optimal size, beyond which error density rises with size. Our study of a large Ada project shows this optimal size to be about 225 lines. Two studies done with other languages elsewhere suggest optimal sizes of 500 and more than 200 lines. Undoubtedly, this optimal size will vary depending on language and style, but we postulate the general phenomenon of falling and then rising error density as related to module size as a universal programming phenomenon.

Based on what is known so far about error density and size, "the smaller the better" is not necessarily good advice to the module designer. On the other hand, extremely long modules should probably also be avoided. The middle road of perhaps 200 to 500 lines, depending on the project, will generally produce the lowest error density and the most maintainable code.

The software tester is well advised to give adequate attention to testing modules of either extreme in size. The software maintainer may want to give more weight to considerations of redesigning and rewriting such modules.

There has been little investigation on either error density or module size. More insight will be gained when data are analyzed by phase and according to error type. Other data, such as time spent in correction and testing and the number of lines changed, could be used to refine the understanding of the relationship between error density and size. There is a notable lack of such data for Ada projects and for projects as large as the one studied here. Future studies will provide a basis for better understanding of the significance of module size in particular and more reliable software in general. ❖

Acknowledgments

I thank John Merritt for directing my attention to this area, Klancy de Nevers for porting the data, Eileen Noel for library searches, Dean and Kent Withrow for reviewing this article, and several anonymous reviewers who made valuable suggestions.

References

1. L. Thomas, *Late-Night Thoughts on Listening to Mahler's Ninth Symphony*, Oxford University Press, Oxford, England, 1983, pp. 143-144.
2. W. Harrison et al., "Applying Software-Complexity Metrics to Program Maintenance," *Computer*, Sept. 1982, pp. 65-79.
3. B. Curtis, S.B. Sheppard, and P. Milliman, "Third-Time Charm: Stronger Prediction of Programmer Preference by Software-Complexity Metrics," *Proc. Fourth Int'l Conf. Software Eng.*, CS Press, Los Alamitos, Calif., 1979, pp. 356-360.
4. D. Kafura and G. Reddy, "The Use of Software-Complexity Metrics in Software Maintenance," *IEEE Trans. Software Eng.*, March 1987, pp. 335-345.
5. J. Martin and C. McClure, *Software Maintenance*, Prentice-Hall, Englewood Cliffs, N.J., 1983, p. 52.
6. T.A. Thayer et al., "Software Reliability Study," Tech. Report RADC-TR-76-238, Rome Air Development Center, Griffiss AFB, N.Y., 1976, pp. 4-51.
7. V. Shen et al., "Identifying Error-Prone Software: An Empirical Study," *IEEE Trans. Software Eng.*, April 1985, pp. 317-324.
8. V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Comm. ACM*, Jan. 1984, pp. 42-52.
9. R.D. Banker and C.F. Kemerer, "Scale Economies in New Software Development," *IEEE Trans. Software Eng.*, Oct. 1989, pp. 1,199-1,205.

Carol Withrow is a principal software engineer at Unisys Communication Systems Division in Salt Lake City. She is the technical leader for the maintenance of the command-and-control software of a military communication system. Before joining Unisys, she was a programmer/analyst with the University of Utah Research Institute with interests in statistics and Earth-science applications. Her interests include software metrics and software-engineering processes.

Withrow has a BS in biology from Arizona State University and an MS in computer science from the University of Utah. She is a member of ACM, ACM SIGSoft, and Computer Professionals for Social Responsibility.

Address questions about this article to Withrow at Unisys Communications, 640 N. 2200 W. St., MS E1D14, Salt Lake City, UT 84116.