23-61

N87 - 24908

R8

An Empirical Study of Software Design Practices

DAVID N. CARD, VICTOR E. CHURCH, AND WILLIAM W. AGRESTI

Abstract—**Software engineers have developed a large body of software design theory and folklore, much of which has never been validated. This paper reports the results of an empirical study of software design practices in one specific environment. The practices examined affect module size, module strength, data coupling, descendant span, unreferenced variables, and software reuse. Measures characteristic of these practices were extracted from 887 Fortran modules developed for five flight dynamics software projects monitored by the Software Engineering Laboratory. The relationship of these measures to cost and fault rate was analyzed using a contingency table procedure. The results show that some recommended design practices, despite their intuitive appeal, are ineffective in this environment, whereas others are very effective.**

Index Terms—**Coupling, fault rate, module cost, reuse, size, Software Engineering Laboratory, strength, unreferenced variables.**

Introduction

SOFTWARE engineers have developed a large body of theory, largely unsupported by quantitative evidence, that specifies the characteristics a good design must incorporate. Formal methodologies, as well as folklore, prescribe practices intended to maximize these characteristics. This paper reports the results of an empirical study of design practices in a Fortran-based scientific computing environment. These practices can be expressed as a set of rules.

- Reuse software wherever possible.
- Keep modules small.
- Include only one function in any module.
- Use parameter rather than COMMON coupling.
- Allow no more than seven descendants to any module.
- Do not create or access data items unnecessarily.

Some important design practices (e.g., information hiding [1] and data abstraction [2]) had to be excluded from this study because they were difficult to measure and/or implement in Fortran. Many factors weigh against simply upgrading the development language: a substantial legacy of previously developed Fortran software, obvious success with the existing language and environment, and lack of a persuasive analysis of the cost effectiveness of alternatives. This study therefore deals only with a small set of design practices likely to be employed in a Fortran-based scientific computing environment.

The purpose of this study was to determine whether or not the observed development cost and fault rate of individual modules were consistent with the purported benefits of the six design practices. Although the term "module" can refer to a more elaborate structure [1], as used in this paper it equates to a Fortran "subroutine" consistent with the Structured Design of Stevens *et al.* [3]. The authors selected three samples of Fortran modules from the Software Engineering Laboratory (SEL) database for study.

Manuscript received April 18, 1985.

The authors are with the System Sciences Division, Computer Sciences Corporation, Silver Spring, MD 20910.

IEEE Log Number 8405411.

Software Engineering Laboratory

The SEL is a research project sponsored by the National Aeronautics and Space Administration and supported by Computer Sciences Corporation and the University of Maryland [4]. The SEL monitors the development of software systems for ground-based spacecraft flight dynamics applications.

The general category of flight dynamics software includes applications that support attitude determination and control, orbit determination and control, and mission analysis. Most of these (primarily Fortran) programs are scientific and mathematical in nature. The attitude systems, in particular, form a large and homogeneous group of software that has been studied extensively. Table I summarizes the characteristics of typical flight dynamics software projects.

The SEL monitors the development of all flight dynamics projects via forms and questionnaires, computer accounting, and a source code analyzer. Programmer hours, errors, and computer use as well as size and complexity measures are recorded. This information is stored on a computer database accessible to all SEL participants. SEL data collection efforts through 1984 include more than 45 flight dynamics projects.

Analysis Approach

Many researchers, notably Boehm [5], have pointed out the difficulty of making reliable statistical inferences based on data collected from production projects. Many uncontrolled (and largely uncontrollable) factors affect the outcome of a real project. DeMarco [6], however, suggests that it is better to use the limited data available than to guess unaided. Hence, the method of this paper was not to pursue a rigorous statistical analysis, but rather to attempt to discover the most visible trends in actual projects.

Although simple statistics are used, the reader should keep in mind that statistics cannot *prove* an assertion; they can only estimate (under the right conditions) the *uncertainty* associated with it. The objective of this paper is not

TABLE I
CHARACTERISTICS OF FLIGHT DYNAMICS SOFTWARE

PROCESS CHARACTERISTICS	AVERAGE	HIGH	LOW
DURATION (MONTHS)	16	21	13
EFFORT (STAFF YEARS)	8	24	2
SIZE (1000 SOURCE LINES OF CODE)			
DEVELOPED	57	142	22
DELIVERED	62	159	33
STAFF (FULL-TIME EQUIVALENT)			
AVERAGE	5	11	2
PEAK	10	24	4
INDIVIDUALS	14	29	7
APPLICATION EXPERIENCE (YEARS)			
MANAGERS	6	7	5
TECHNICAL STAFF	4	5	3
OVERALL EXPERIENCE (YEARS)			
MANAGERS	10	14	8
TECHNICAL STAFF	9	11	7

NOTES: TYPE OF SOFTWARE: SCIENTIFIC, GROUND-BASED, INTERACTIVE GRAPHIC.
LANGUAGES: 85 PERCENT FORTRAN, 15 PERCENT ASSEMBLER MACROS.
COMPUTERS: IBM MAINFRAMES.

TABLE II
DATA SAMPLES STUDIED

SAMPLE	NUMBER OF MODULES	SOFTWARE ORIGIN	COST[a] MINIMUM
A	887	ALL	NO
B	453	NEW ONLY	YES
C	434	NEW ONLY	NO

[a]MODULES REQUIRING LESS THAN 1 HOUR OF EFFORT EXCLUDED.

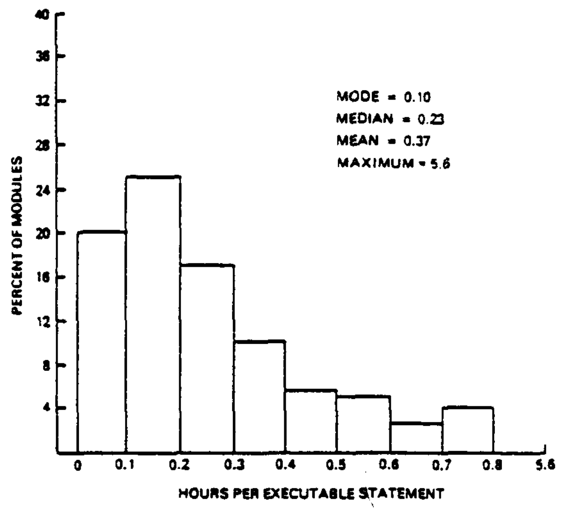


Fig. 1. Distribution of development cost (for sample B).

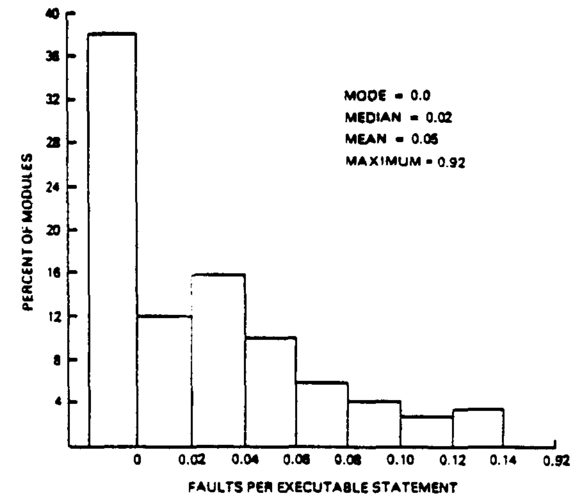


Fig. 2. Distribution of fault rate (for sample B).

to pass final judgment on any design practice. It is hoped rather that the presentation of this empirical data will provide additional information to researchers and practitioners evaluating alternative design strategies.

The evaluation of each design practice is based on an analysis of one of three data samples extracted from five projects monitored by the SEL. Table II describes the three samples. Although all modules in these samples come from the same projects, the samples differ with respect to the selection criteria and measures included for study. Sample A includes all Fortran modules from the five projects for which the initial set of measures was complete. Sample B is a subset of A consisting only of newly developed nontrivial modules. All of the design practices under consideration could not, however, be evaluated with the measures available in samples A and B. Sample C includes additional measures extracted by a special design analysis tool. Only three of the five projects have been processed by the design analysis tool as of this date.

Examination of module cost and fault rates provided the basis for the evaluation of the design practices. *Module cost* includes the programmer hours of effort spent designing, coding, and unit/integration testing the module. Dividing by the number of executable statements normalizes this measure with respect to module size. Faults were counted for each module from the completion of unit/integration testing until the end of acceptance testing. Dividing the number of faults by the number of executable statements produces the *fault rate*. The software studied remains operational for a relatively short period (seldom more than five years), and consequently, maintenance is minimal. Faults and costs are not recorded for individual modules during this period.

An initial examination of the data revealed that neither module cost nor fault rate was normally distributed; Figs. 1 and 2 illustrate this for sample B. Consequently, the authors adopted a contingency table approach to the analysis rather than relying on normal-distribution-based techniques such as regression and analysis of variance. To perform the contingency table analysis, every module was assigned to one of three ordered classes (of nearly equal size) for each of the quality measures of development cost (low, medium, high) and fault rate (zero, medium, high).

The values 0.151 and 0.322 programmer hours per executable statement divided the modules into the three cost classes (i.e., 0.151 or less was low cost). The value 0.045 faults per executable statement distinguished between medium- and high-fault-rate classes. One class consisted of those modules with no reported faults.

Similarly, ordered classes of conformance to the design practices were defined. The strength of the relationships between the ordered classes of design characteristics and quality measures was assessed by calculating the gamma (γ) correlation statistic [7]. This statistic varies from -1.0 to $+1.0$.

Design Practices and Characteristics

Characteristic measures were defined for each of the six design practices. Each characteristic was studied using the contingency table approach. The results of those analyses are presented here.

Software Reuse

Although not always recognized as such, an important design decision involves the reuse of existing software. Some good ideas about how to write reusable software have been published [8]; however, opportunities to reuse software must be recognized in the design activity. The goals of this analysis (based on sample A) were to identify the types of software that are reused in the flight dynamics environment and to quantify the benefits of software reuse.

Table III lists some of the characteristics of software reuse. Executable statements measure module size. Mean decisions measure module complexity. The table indicates that the modules that are reused without modification (old) tend to be small and simple (exhibiting a relatively low decision rate). A more detailed cross-classification (not shown) revealed that 55 percent of all old modules are high-strength algorithmic modules, the type likely to be found in mathematical software libraries. Table III shows that extensively modified modules tended to be the largest in terms of the number of executable statements.

Tables IV and V clearly demonstrate the cost and quality benefits of software reuse. Fully 98 percent of old modules proved to be fault free, and 82 percent of them fell into the lowest cost (per executable statement) category. Significant (γ) correlations are associated with both of these relationships. (Percentages do not add to exactly 100 due to rounding.) These results are consistent with previous SEL studies of reused code [4], which indicated that reusing a line of code costs only 20 percent of the cost of developing it new. Because these four classes of software differ substantially with respect to structure and quality measures, the subsequent analyses are based on new modules only.

Module Size

The 453 modules in sample B were classified into three approximately equal ordered groups on the basis of the number of executable statements in each module. Table VI shows the results of this classification.

TABLE III
Characteristics of Reused Software

SOFTWARE TYPE	NUMBER OF MODULES	EXECUTABLE STATEMENTS	MEAN DECISIONS PER EXECUTABLE STATEMENT[a]
NEW	532	55	0.30
EXTENSIVELY MODIFIED	132	83	0.29
SLIGHTLY MODIFIED	163	51	0.25
OLD (UNCHANGED)	60	28	0.20

[a]NUMBER OF LOGICAL DECISIONS DIVIDED BY NUMBER OF EXECUTABLE STATEMENTS.

TABLE IV
Software Reuse and Development

SOFTWARE TYPE	DEVELOPMENT COST (PERCENT)		
	LOW	MEDIUM	HIGH
NEW	42	28	30
EXTENSIVELY MODIFIED	52	27	22
SLIGHTLY MODIFIED	63	21	17
OLD (UNCHANGED)	82	8	10

NOTE: GAMMA (γ) = -0.33; PROBABILITY THAT $\gamma = 0$ IS LESS THAN 0.001.

TABLE V
Software Reuse and Fault Rate

SOFTWARE TYPE	FAULT RATE (PERCENT)		
	ZERO	MEDIUM	HIGH
NEW	44	29	27
EXTENSIVELY MODIFIED	52	33	15
SLIGHTLY MODIFIED	69	17	14
OLD (UNCHANGED)	98	0	2

NOTE: GAMMA (γ) = -0.43; PROBABILITY THAT $\gamma = 0$ IS LESS THAN 0.001.

TABLE VI
Module Size Distribution

MODULE SIZE	NUMBER OF FORTRAN MODULES	EXECUTABLE STATEMENTS	MEAN DECISIONS PER EXECUTABLE STATEMENT
SMALL	154	1 TO 31	0.31
MEDIUM	148	32 TO 64	0.31
LARGE	151	65 OR MORE	0.32

The largest module in the sample contained 267 executable statements. The dividing line of 31 executable statements is significant because, in the environment studied, it corresponds to about 60 source lines of code. Many programming standards [9] limit module size to one page (or 50–60 source lines of code); one purpose of the study was to test the validity of such standards.

A cross-tabulation of module size with development cost showed a correlation (γ) of -0.31. The probability of this

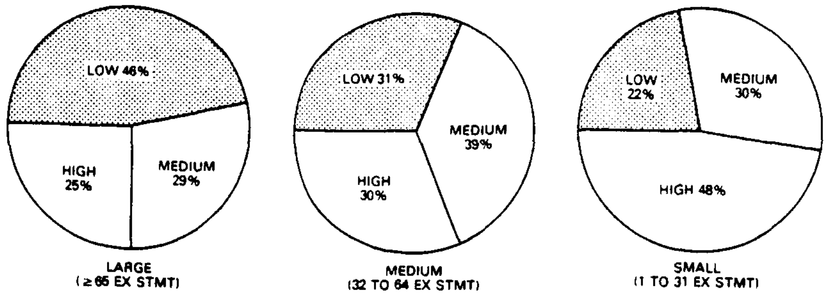


Fig. 3. Development cost for classes of module size.

correlation being due to random factors is less than 0.001. Although the magnitude of this correlation seems small, Fig. 3 provides a better illustration of its importance. As the figure indicates, fully 46 percent of large modules fell into the lowest cost class, whereas just 22 percent of small modules were rated as low cost.

No significant relationship was found between module size and fault rate. Two recent studies [10], [11] concluded that smaller modules were more fault prone. These studies, however, adopted parametric approaches to the statistical analysis. Sixty percent of the small modules in sample B contained no faults. Nevertheless, this size class exhibited the highest *average* fault rate because a small module with even a single fault will show a very high fault rate. That is the phenomenon detected by Basili and Perricone [10] and Shen *et al.* [11].

The effects of programmer performance and the possibility of an interaction between module size and module strength were subsequently considered in a more detailed analysis of these data [12]. That consideration did not change the conclusion that larger modules cost less to develop (per executable statement) than small ones. A similar result has been reported for another class of software [13].

Module Strength

Myers [14] defines seven levels of module strength. In descending order, these are functional, informational, communicational, procedural, classical, logical, and coincidental. A high- (functional) strength module performs a single well-defined function. Myers contends that high-strength modules are superior to low-strength modules. Although it was not possible to test this theory exactly, a reasonable approximation was made. Some recent attempts to develop objective measures of module strength [15], [16] seem promising, but are not (in their present forms) easily applied; consequently, they were not employed in this study. An earlier study [17] based on objective but indirect measures of module strength (cohesion) proved inconclusive.

Programmers determined the strength of a module via a checklist, rating each module they developed as performing one or more of the following functions: input/output, logic/control, and/or algorithmic processing. Distinguishing the *types* of functions seemed to be a less ambiguous task than identifying the *number* of functions because the number of functions depends on the level of decomposition recognized by the respondent. Those modules described as having only one function were classified as high strength; those having two functions, medium strength; and those having three or more functions, low strength. Table VII summarizes the results of this classification process (for sample B).

TABLE VII
MODULE STRENGTH DISTRIBUTION

MODULE STRENGTH	NUMBER OF FORTRAN MODULES	MEAN EXECUTABLE STATEMENTS	MEAN DECISIONS PER EXECUTABLE STATEMENT
LOW	90	77	0.29
MEDIUM	176	60	0.32
HIGH	187	48	0.32

A cross-tabulation of module strength with fault rate showed a correlation (γ) of −0.35. The probability that this correlation is due to random factors is less than 0.001. Again, a figure provides a better indication of the magnitude of this correlation. Fifty percent of high-strength modules were fault free, whereas only 18 percent of low-strength modules were fault free (Fig. 4).

No significant relationship was discovered between module strength and development cost. The effects of programmer performance and the possibility of an interaction between module size and module strength were subsequently considered in a more detailed analysis of these data [12]. The conclusion, however, remained unchanged: developing high-strength modules is good practice.

Data Coupling

There are two ways that Fortran modules can be coupled directly: through calling sequence parameters or through COMMON block variables. (A COMMON block is a global data area.) Some authors have argued against COMMON coupling [9] even if, as a result, calling sequences become long and unwieldy. A design measure was devised to evaluate that argument.

For this analysis, the modules in sample C were grouped into three ordered classes with respect to the percentage of *referenced* input/output variables in COMMON: zero, ≤ 15 percent, and > 15 percent. Separate analyses were performed for each of three classes of modules based on

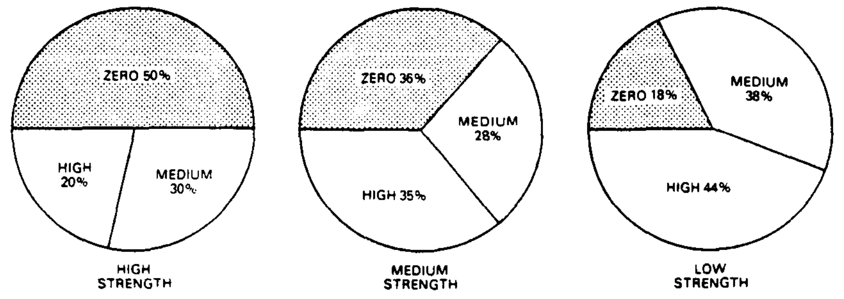


Fig. 4. Fault rate for classes of module strength.

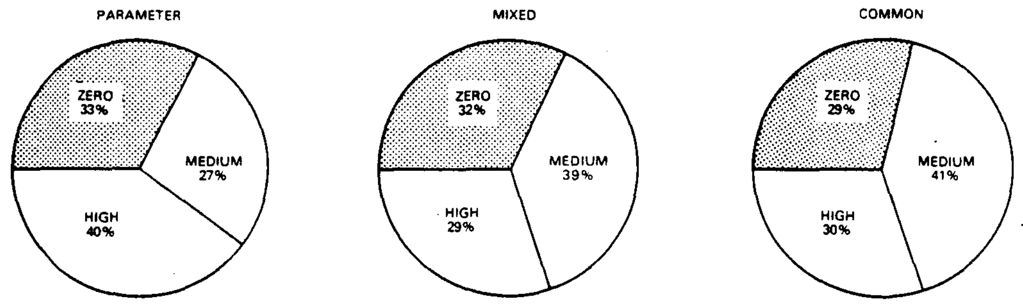


Fig. 5. Fault rate by coupling type.

TABLE VIII
EFFECTS OF COUPLING TIME

MODULE CLASS[a]	MODULE COUNT	CORRELATION[b] OF PERCENT COMMON COUPLING WITH	
		FAULTS	COST
NONTERMINAL	311	−0.06	−0.15
TERMINAL	123	−0.04	−0.28
UTILITIES	76	−0.12	−0.41[c]

[a]LOCATION IN DESIGN STRUCTURE.
[b]GAMMA (γ) STATISTIC.
[c]PROBABILITY <0.01 THAT γ ACTUALLY IS ZERO.

TABLE IX
CHARACTERISTICS OF MODULE INVOCATION

INVOCATION CLASS[a]	MODULE COUNT	MEAN EXECUTABLE STATEMENTS	MEAN CALLS[b]
ZERO[c]	123	36	1.1
ONE	81	45	3.0
TWO TO SEVEN	187	80	6.5
MORE THAN SEVEN	43	88	14.8

[a]BASED ON CALLS TO APPLICATION SOFTWARE ONLY.
[b]INCLUDES ALL CALLS OF ANY TYPE.
[c]TERMINAL NODES EXCLUDED FROM ANALYSIS.

position in the software structure: nonterminal nodes (fan-out >0), terminal nodes (fan-out = 0), and utilities (fan-in > 1). As Table VIII indicates, no relationship was observed between fault rate and coupling.

Fig. 5 illustrates this; the percentage of zero-fault modules is about the same for both parameter and COMMON coupled modules. On the other hand, Table VIII also indicates that, for utility modules, a cost savings is associated with COMMON coupling (−0.41 correlation with cost). Earlier recommendations that common coupling was best avoided may have been based on experience before the general availability of "INCLUDE" processors. In an environment where only a single version of a COMMON block definition needs to be maintained, COMMON coupling is an acceptable, and sometimes preferable, alternative to parameter coupling. Furthermore, another study [18] failed to show any significant difference between global and parameter coupling with respect to modifiability.

Descendant Span

Another basic design principle is that no module should call too many (i.e., more than seven) other modules. Furthermore, a module that calls only one other module might just as well include the other module's function within itself. One formulation of this concept is an adaptation of the "7 ± 2 rule" [19], which states that each module should call from five to nine other modules, except in the case of terminal nodes. This rule is a formal element of the System Activity Modeling Method [20].

For this analysis, the modules in sample C were grouped into three ordered classes with respect to the number of descendants: one, two to seven, and more than seven. (Terminal nodes were not included in this analysis.) Table IX shows some characteristics of these classes. The results of cross-classification indicate that modules with more descendants tend to cost more (per executable statement) to develop (γ = 0.25) and have a higher fault rate

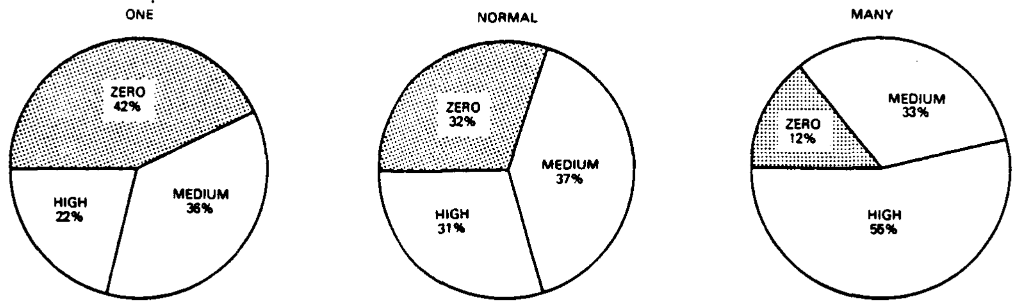


Fig. 6. Fault rate by descendant count.

(γ = 0.33). The probability of these correlations being due to chance is less than 0.01.

Fig. 6 illustrates the magnitude of the difference among classes for fault rate. Only 12 percent of modules with more than seven descendants were fault free, whereas 42 percent of modules with only one descendant were fault free. Apparently, the simpler the invocation structure in a module, the better. This measure of structural complexity has proven to be clearly related to *fault rate* (faults per executable statement), whereas more complex measures based on counts of decisions and operators have not [21]. On the other hand, the total *number of faults* appears to increase with size, decisions, operators, etc. [21]. Earlier SEL studies dealing with descendant span (e.g., [22]) were handicapped because the source analyzer program that produced this measure did not distinguish between calls to system/library routines and calls to other application modules. The design analysis tool developed for this study remedies that deficiency.

Unreferenced Variables

Most guidelines encourage designers (and programmers) not to create or access data items unnecessarily. Failure to reference a variable (in the completed module) indicates that its presence is unnecessary. Unreferenced variables in a Fortran module arise from three sources: locally unused variables carried along in COMMON, unused variables defined locally, and unreferenced calling sequence parameters. In this environment, the presence of unreferenced variables in COMMON usually indicates that the COMMON block represents a data structure (e.g., record, state definition) rather than a simple subsitute for calling sequence parameters.

For these analyses, the modules in sample C were grouped into three ordered classes with respect to the percentage of unreferenced variables. Separate analyses were performed for locally or calling sequence defined variables and for variables accessible in COMMON. The percentage of unreferenced variables of each type was calculated for each module. Then the modules were classified as having none, a medium percentage, or a high percentage of unreferenced variables. Table X lists the results obtained by comparing the presence of unreferenced variables to cost and fault rate.

TABLE X
EFFECTS OF UNREFERENCED VARIABLES

LOCATION OF VARIABLES	MODULE COUNT	CORRELATION[a] OF PERCENT UNREFERENCED VARIABLES WITH	
		FAULTS	COST
COMMON	241	-0.33^b	-0.03
LOCAL AND CALLING SEQUENCE	434	0.44^b	0.29^b

[a]GAMMA (γ) STATISTIC.
[b]PROBABILITY <0.01 THAT γ ACTUALLY IS ZERO.

These results indicate that keeping logically related variables together (in COMMON), even if some are not used, is associated with a lower fault rate. (Possibly the additional contextual information provided by the data structure promotes the correct use of the data items needed. This effect may be related to the concept of data abstraction [2].) Table X further implies that a high proportion of unreferenced local and calling sequence variables signifies sloppy workmanship, leading to high cost and fault rates. Fig. 7 illustrates the magnitude of this association. Only 17 percent of modules with more than 20 percent unreferenced (local and calling sequence) variables were fault free, whereas 46 percent of modules with no such unreferenced variables were fault free. This result provides indirect support for the use of information hiding [1]. That is, the presence of unreferenced variables in the calling sequence suggests a lack of information hiding.

CONCLUSIONS

Overall, these results suggest that although many assumptions about system design and software development are well founded some need to be rethought. Changes in computer hardware and other software technology can alter the effectiveness of a design practice (e.g., COMMON coupling). The specific results obtained from this study can be summarized as follows.

- Arbitrary module size limitations can increase module cost.
- High module strength reduces fault rate.
- COMMON coupling reduces development cost for utility modules.

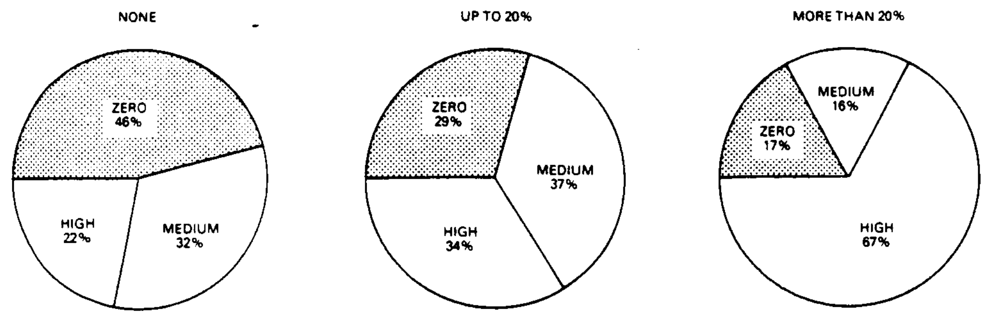


Fig. 7. Fault rate by unreferenced variables.

- Modules with many descendants are more fault prone than those with few.
- Modules with many (non-COMMON) unreferenced variables are more fault prone and ultimately cost more than those with few.
- Software reuse reduces system cost and fault rate.

Empirical studies such as this provide a mechanism for verifying the relevance of items in the store of software engineering knowledge. All design practices are not equally effective in all environments. In every case, the effects of technology must be measured before practitioners can make informed decisions. These specific results probably apply to any similar scientific software development environment. Despite advances in other programming languages, Fortran seems likely to continue to be used for many scientific applications [23]. Thus, studies of current Fortran practice are important and should be encouraged.

Acknowledgment

The authors would like to thank F. E. McGarry, G. T. Page, and V. R. Basili for their support in this work, and the referees for their assistance in polishing this presentation.

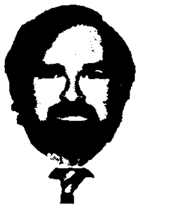
References

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053-1058, Dec. 1972.
[2] B. Liskov and S. Zilles, "Specification techniques for data abstractions," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 7-19, 1975.
[3] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115-139, 1974.
[4] D. N. Card, F. E. McGarry, G. T. Page *et al.*, *The Software Engineering Laboratory*, NASA/GSFC, Feb. 1982.
[5] B. W. Boehm, *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
[6] T. DeMarco, *Controlling Software Projects*. New York: Yourdon Press, 1982.
[7] L. A. Marascuilo and M. McSweeney, *Nonparametric and Distribution Free Methods for the Social Sciences*. California: Brooks/Cole, 1977.
[8] B. Meyer, "Principles of package design," *Commun. ACM*, vol. 25, pp. 385-395, 1982.
[9] B. W. Kernighan and P. S. Plauger, *The Elements of Programming Style*. New York: McGraw-Hill, 1974.
[10] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Commun. ACM*, vol. 27, pp. 42-52, 1984.
[11] V. Y. Shen, T. Yu, S. M. Thebaut, and L. R. Paulsen, "Identifying error-prone software—An empirical study," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 317-324, 1985.
[12] D. N. Card, G. T. Page, F. E. McGarry, "Criteria for software modularization," in *Proc. Eighth Int. Conf. Software Eng.*, 1985.
[13] P. C. Belford, R. C. Berg, and T. L. Hannan, "Central flow control software development: A case study of the effectiveness of software engineering techniques," in *Proc. Fourth Int. Conf. Software Eng.*, 1979, pp. 85-93.
[14] G. J. Myers, *Composite/Structured Design*. New York: Van Nostrand-Reinhold, 1978.
[15] R. D. Cruickshank and J. E. Gaffney, "Measuring the development process: Software design coupling and strength matrices," in *Proc. Fifth Annu. Software Eng. Workshop*, NASA/GSFC, Nov. 1980.
[16] T. J. Emerson, "A discriminant metric for module cohesion," *Proc. Seventh Int. Conf. Software Eng.*, 1984, pp. 294-303.
[17] D. A. Troy and S. H. Zweben, "Measuring the quality of structured designs," *J. Syst. Software*, vol. 2, pp. 113-120, 1981.
[18] J. B. Lohse and S. H. Zweben, "Experimental evaluation of software design principles: An investigation into the effect of module coupling on system modifiability," *J. Syst. Software*, vol. 4, pp. 301-308, 1984.
[19] G. A. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," *Psychol. Rev.*, vol. 63, pp. 81-97, 1956.
[20] *SAMM (System Activity Modeling Method) Primer*, Boeing Computer Services Co., Doc. BCS-10167, 1978.
[21] F. E. McGarry, "Measuring software technology," in *Proc. Seventh Annu. Software Eng. Workshop*, NASA/GSFC, Dec. 1982.
[22] V. R. Basili, R. W. Selby, and T. Phillips, "Metric analysis and data validation across FORTRAN projects," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 652-663, 1983.
[23] M. Metcalf, "Has FORTRAN a future?," presented at the Europhys. Conf. Software Eng. Methods, Tools Comput. Phys., Aug. 1984.

David N. Card received the B.S. degree in interdisciplinary studies from American University, Washington, DC, in 1975 and has since done additional graduate study in applied statistics.

For the past six years he has been a member of the Computer Sciences Corporation (CSC) team supporting the Software Engineering Laboratory (SEL). His areas of activity in the SEL include software measurement, technology evaluation, cost estimation, and standards development.

Mr. Card is a member of the Association for Computing Machinery and the American Statistical Association.



Victor E. Church received the B.S. degree from the University of Maryland, College Park, in 1973 and the M.S. degree in computer science and physiology from George Washington University, Washington, DC, in 1977.

He is a programming and research manager at Computer Sciences Corporation (CSC). Since 1979 he has worked at CSC in the areas of software design and software engineering research. Previously, at the GWU Medical School he studied student computer interaction while managing a computer-based educational facility.

Mr. Church is a member of the IEEE Computer Society and the Association for Computing Machinery.

William W. Agresti received the B.S. degree from Case Western Reserve University, Cleveland, OH, and the M.S. and Ph.D. degrees from New York University, New York, NY.

He is a Senior Computer Scientist with Computer Sciences Corporation (CSC) where his applied research and development projects support the Software Engineering Laboratory (SEL) at NASA's Goddard Space Flight Center. His research interests are in software process engineering and software metrics. From 1973 to 1983 he held various faculty and administrative positions at the University of Michigan, Dearborn, including founding Director of Computer and Information Sciences, Associate Professor of Industrial and Systems Engineering, and Associate Dean of the School of Engineering. During this period, he was a consultant to Ford Motor Company and other corporations and agencies, and he was a founder of the software products firm A.I.S. International, Inc.

Dr. Agresti is a member of the IEEE Computer Society and the Association for Computing Machinery, serving as President of the Metropolitan Detroit Chapter of ACM in 1979–1980.