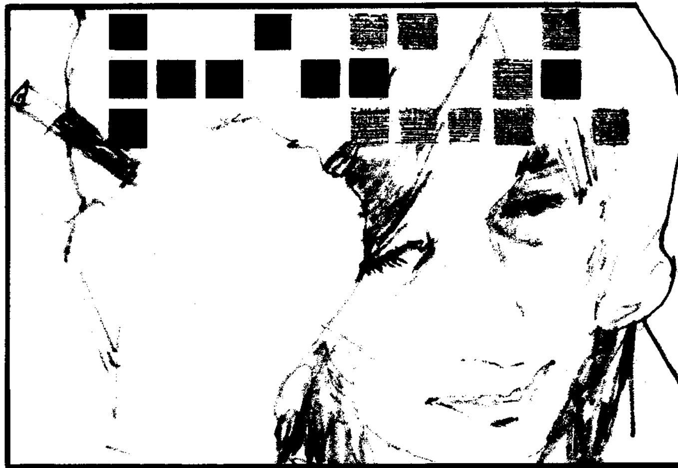
The effects of human factors on "high-level" software properties—too intangible to quantify directly—can be inferred from the collective behavior of related "low-level" aspects.

An Investigation of Human Factors in Software Development

Victor R. Basili
Robert W. Reiter, Jr.
University of Maryland



Recently, considerable attention has been devoted to the notion that factors directly related to the psychological nature of human beings play a major role in the development of computer software.[1-10] If human factors do significantly affect software development, then varying the size of the programming team and the degree of methodological discipline—two supposedly potent human factors—should induce measurable differences in both the development *process* and the developed *product*. Controlled experimentation involving easily measured "low-level" programming aspects can serve not only to verify this hypothesis but also to accumulate a detailed set of empirically supported conclusions. Interpreting these conclusions in view of subjective reasoning about software can yield further understanding about the effect of human factors on certain "high-level" software properties that are difficult to measure directly.

To this end, we conducted a controlled experiment involving several replications of a specific software development task under varying programming environments. For each replication, successive versions of the software were entered in a historical data bank that recorded details of the development process and product. A host of low-level measurements, such as the number of textual revisions during development and the number of decision statements in the delivered source code, were extracted from the data bank and statistically analyzed. Some of these measuremments were "confirmatory," planned in advance and expected to show differences among the programming environments; but many were simply "exploratory." For each measured aspect, differences in both the *expectancy* and the *predictability* of its behavior under the different programming environments were checked for statistical significance. The experiment's conclusions were interpreted according to subjective reasoning about software in order to perceive their implications regarding certain high-level software properties, such as reliability and complexity.

The investigation focused on the effects of two particular human factors: the size of the programming "team" deployed and the degree of methodological discipline employed. For the team-size factor, individual programmers working alone were compared to teams of three programmers working together. For the degree-of-discipline factor, an ad hoc approach allowing programmers to develop software without externally imposed methodological constraints was compared to a disciplined methodology obliging programmers to follow certain modern programming practices and team organization procedures. It should be noted that the terms "methodology" and "methodological" are used herein to connote a comprehensive integrated set of development techniques as well as team organizations, rather than a particular technique or organization in isolation.

Two distinguishing characteristics of this experimental investigation deserve emphasis. First, this study was based upon direct quantification of software development phenomena: measurement that is quantitative (on at least an interval scale[11]), objective (without inaccuracy due to human subjectivity), unobtrusive (to those developing the software), and automatable (not dependent on human agency). Second, this study was concerned with controlled experimentation involving an entire software development project of nontrivial size in a lifelike setting. This particular experiment represents a reasonable compromise between "toy" experiments, which

0018-9162/79/1200-0021$00.75 © 1979 IEEE

often suffer from artificiality or lack practical application, and "production" experiments, which incur prohibitively high costs.

This paper discusses human factors issues relative to initial results from an extensive software engineering research project; a comprehensive treatment of the overall study appears elsewhere.[12-14]

Designing and conducting the experiment

Experimental design. Completion of a specific software development project was the basic task involved in the experiment. Nineteen replications of the basic task were performed concurrently and independently by separate software development teams under controlled but varied conditions. Two programming factors (each with two levels) were selected as the independent variables: size of programming team (single individual, three-person team) and degree of methodological discipline (ad hoc approach, disciplined methodology). Three experimental treatment groups, designated AI, AT, and DT (containing 6, 6, and 7 teams, respectively), operated under a different factor-level combination:

AI —ad hoc approach, single individual;
AT —ad hoc approach, three-person team;
DT —disciplined methodology, three-person team.

There was no DI group since the disciplined methodology cannot be exercised by an individual programmer.

The dependent variables to be observed consisted of a large set (over 130) of quantifiable programming aspects characterizing the development process and the developed product. Following a reductionist approach, many programming factors were held constant across all programming teams: programming project, project specifications, implementation language, calendar schedule, computer resource allocation, and automated debugging tools. However, two other factors could not be explicitly controlled and were allowed to vary among the teams: the personal ability and/or experience of the participants and the amount of time and/or effort they actually devoted to the project.

Software development methodologies. The ad hoc approach exercised by groups AI and AT allowed programmers to develop software in a manner entirely of their own choosing. No methodology was being taught in the course that these subjects were taking. The disciplined methodology exercised by group DT consisted of an integrated set of state-of-the-art techniques, including top-down design, process design language (PDL), functional expansion, design and code reading, walk-throughs, and chief programmer team organization. These techniques were being taught as an integral part of the course that these subjects were taking, using texts by Linger, Mills, and Witt,[15] Basili and Baker,[16] and Brooks.[17] Since the subjects were novices in the methodology, they executed the techniques to varying degrees of thoroughness and were not always as successful as seasoned users of the methodology would be.

Experimental setting. The experiment was conducted during the spring 1976 semester in conjunction with regular academic courses at the University of Maryland. A graduate-level academic setting provided the opportunity both to achieve an adequate experimental design and to simulate key elements of a production environment. The experimental task and treatments were built into the course material and assignments of two comparable advanced elective courses. Everyone in the two classes participated in the experiment; they were aware of being monitored, but had no knowledge of what was being observed or why.

The programming application was a simple compiler, involving string processing and translation from an Algol-like language to a zero-address machine code. The total task was to design, implement, test, and debug a complete system from given specifications. The scope of the project excluded both extensive error handling and user documentation. The project was of modest but nonnegligible difficulty, requiring roughly a two man-month effort and resulting in systems that averaged over 1200 lines of high-level language source code. All facets of the project itself were fixed and uniform across all development teams.

The participants were advanced undergraduate and graduate students in the Department of Computer Science. None were novice programmers, all had completed at least four semesters of programming course work, several were about to graduate and take programming jobs in government or industry, and a few had as much as three years of professional programming experience. Generally speaking, they were all familiar with both the implementation language and the host computer system, but inexperienced in team programming and the disciplined methodology.

It was necessary in the statistical model to assume homogeneity among the participants with respect to personal factors such as ability and/or experience, motivation, time and/or effort devoted to the project, etc. As a reasonable measure of individual programmer skill levels under the circumstances of this study, the participants' grades from a pertinent prerequisite course provided a post-experimental confirmation of at least one facet of this assumed homogeneity: the distribution of these grades among the three experimental groups would have displayed the same degree of homogeneity as was actually observed in over 9 out of 10 purely random assignments of the participants to the groups. If anything, based on the researchers' subjective judgement, the participants in group AI seemed to have a slight edge over those in groups AT and DT with respect to native programming ability, while groups AI and AT seemed slightly favored over group DT with respect to formal training in the application area.

The implementation language was the high-level, structured-programming language SIMPL-T,[18]

which is taught and used extensively in course work at the university. SIMPL-T contains the following control constructs: sequence, IFTHEN, IFTHEN-ELSE, CASE, WHILEDO, EXIT from loop, and RETURN from routine (but no GOTO). SIMPL-T allows two levels of data declaration scope, local to an individual routine or global across several routines, but routines may not be nested. The language adheres to a philosophy of "strong data typing" and supports integer, character, and string data types and single-dimension array data structures. It provides the programmer with both recursion and string-processing capabilities similar to PL/I.

Data collection and reduction. For the experiment, data collection was automated on-line, followed by an off-line data reduction step. While the software projects were being developed, the computer activities of each team were automatically and unobtrusively monitored. Special module compilation and program execution processors (invoked by very slight changes to the regular command language) created a historical data bank, consisting of all source code and test data accumulated throughout the project development period, for each development team. The raw information in this data bank was subsequently reduced to obtain the experimental observations. Scores for each low-level programming aspect were extracted directly and algorithmically from the data bank, thus ensuring their objectivity.

Programming aspects measured. The low-level programming aspects are specific isolatable and observable features of programming phenomena, related to either the process or the product of software development. Process aspects represent characteristics of the development process itself, in particular, the cost and required effort as reflected in the number of computer job steps (or runs) and the amount of textual revision of source code during development. Product aspects represent characteristics of the final product that was developed, in particular, the syntactic content and organization of the symbolic source code. Examples of product aspects are number of lines, frequency of particular statement types, and average size of data variables' scope. The range of features considered as product aspects includes control-flow constructs, data variable organization, and inter-routine communication. For each programming aspect there exists an associated metric, a specific algorithm which ultimately defines and measures that aspect. (A complete list of programming aspects and explanatory notes appear in the appendices.)

Confirmatory. The complete set of programming aspects may be partitioned into two subsets based upon the motivation for their inclusion in the study. Several aspects, denoted as confirmatory, had been consciously planned in advance of collecting and extracting data, because intuition suggested that they would serve particularly well as quantitative indicators of important qualitative characteristics of software development phenomena. It was hypothesized a priori that certain differences among the groups would be indicated by the confirmatory aspects, as detailed elsewhere.[14]

Exploratory. The remaining aspects, denoted as exploratory, were considered mainly because they could be collected and extracted cheaply along with the confirmatory aspects. There was little serious expectation that these exploratory aspects would be useful indicators of differences among the groups, but they were included in the study with the intent of observing as many aspects as possible on the off chance of discovering any unexpected tendency or difference. Thus, the study combines elements of both confirmatory and exploratory data analysis within one common experimental setting.[19]

It should be noted that a large percentage of the product aspects fall into the exploratory category. On the whole, the examined product aspects represent a fairly extensive taxonomy of the surface features of software. The idea that important software qualities (e.g., complexity) could be measured by counting such surface features has generally been disregarded as too simplistic.[20] A resolve to study these surface features empirically, to see if something might turn up, before rejecting the underlying idea, was partially responsible for their inclusion in the study.

Objective results and interpretations

The study's objective results are the statistical conclusions for the programming aspects considered in the experiment; they are reported in Tables 1 through 9. The study's interpretations state general trends in the conclusions based on classifications reflecting certain abstract programming notions, such as cost, modularity, and data organization.

The tables express each statistical conclusion in the concise form of a three-way comparison outcome "equation." It states any observed differences, and the directions thereof, among the programming environments represented by the three groups examined in the study: ad hoc individuals, AI; ad hoc teams, AT; and disciplined teams, DT. The equality AI = AT = DT expresses the null outcome of no systematic difference among the groups. (Note that within the tables a simple pair of equal signs appears in place of the null outcome.) An inequality, e.g., AI < AT = DT or DT < AI < AT, expresses a non-null (or alternative) outcome of certain systematic differences among the groups in stated directions. A critical level (or risk) value is also associated with each non-null outcome, indicating its individual reliability. This value is the probability of having erroneously rejected the null conclusion in favor of the alternative; it also provides a relative index of how pronounced the differences were in the sample data. If no alternative outcome could be supported at a critical level below 0.20, the comparison defaulted to the null outcome.

Two kinds of comparisons, location and dispersion, were made for each measured aspect. Location comparisons deal with a measure's central tendency or

average value; dispersion comparisons deal with a measure's variability around its central tendency. Note that examination of both location and dispersion differences among the three groups imposes a certain duality on the entire investigation, since it addresses both the expectancy and the predictability of software development behavior under different programming environments.

Regardless of the particular aspects being measured and the particular kinds of comparisons being made, it is important to understand the implications of the comparison outcome equations just in terms of the experimental factors. The outcome $AI \neq AT = DT$ (i.e., either $AI < AT = DT$ or $AT = DT < AI$) indicates a difference attributable to the team size factor alone; the measured aspect is apparently sensitive to differences in team size but not to differences in methodological discipline. Similarly, the outcome $DT \neq AI = AT$ (i.e., either $DT < AI = AT$ or $AI = AT < DT$) indicates a difference attributable to the methodological discipline factor alone.

However, the outcome $AT \neq DT = AI$ (i.e., either $AT < DT = AI$ or $DT = AI < AT$) is not as easy to explain. Such an outcome clearly indicates an interaction between the levels of the two experimental factors, as though the effect (on the measured aspect) of an increase in team size were somehow counteracted by an accompanying increase in methodological discipline. It is the researchers' belief, as elaborated elsewhere,[12,14] that the $AT \neq AI = DT$ outcome indicates a difference attributable to an intangible "conceptual integrity" factor that is, in turn, predictably sensitive to the tangible factors of team size and methodological discipline.

In general, interpretations express the researchers' own ideas, according to personal intution about programming and software, of the implication and importance of the study's objective results. The interpretations presented here are based on a simple classification of the low-level programming aspects. Each class consists of aspects that are related by some common feature (e.g., all aspects relating to the program's statements, statement types, and statement nesting), and classes are not necessarily disjoint (i.e., a given aspect may be included in two or more classes). A particular high-level software property (in the example, control structure organization) is associated with each class. Such a classification provides a framework for jointly interpreting the corresponding statistical conclusions in light of the underlying issues by which the aspects themselves are related.

The programming aspects measured in this study were categorized into a hierarchy of nine classes (with about 10 percent overlap overall), outlined as follows:

High-Level Software Property	*Class*
Development Process Efficiency	
Effort (job steps)	I
Errors (program changes)	II
Final Product Quality	
Gross Size	III
Control-Construct Stucture	IV
Data Variable Organization	V
Modularity	
Packaging Structure	VI
Invocation Organization	VII
Inter-Routine Communication	
Via Parameters	VIII
Via Global Variables	IX

Tables 1 through 9 list the individual aspects comprising each class, together with the corresponding statistical conclusions for both location and dispersion comparisons. For each aspect class, the following interpretations were formulated by jointly examining the particular statistical conclusions and intuitivley abstracting the empirically derived details, in order to glimpse something of how the corresponding high-level software property is affected by the team size and methodological discipline factors.

Class I: Effort (job steps). Within Class I, process aspects dealing with computer job steps, there is strong evidence that the teams using the disciplined methodology reduced their average development costs relative to both the other teams and the individuals (see Table 1). As a class, these aspects directly reflect the frequency of computer system operations (i.e., module compilations and test program executions) during development. They are one possible way of measuring machine costs, in units of basic operations rather than monetary charges.

Table 1.
Conclusions for Class I: Effort (job steps).

PROGRAMMING ASPECT	LOCATION COMPARISON OUTCOME**	LOCATION CRITICAL LEVEL	DISPERSION COMPARISON OUTCOME**	DISPERSION CRITICAL LEVEL
* COMPUTER JOB STEPS	DT < AI = AT	0.003	= =	
* MODULE COMPILATION	DT < AI = AT	0.022	= =	
* UNIQUE	DT < AI = AT	0.011	= =	
IDENTICAL	= =		= =	
* PROGRAM EXECUTION	DT < AI = AT	0.022	= =	
MISCELLANEOUS	DT < AI = AT	0.144	AT = DT < AI	0.077
* ESSENTIAL	DT < AI = AT	0.003	= =	
AVERAGE UNIQUE COMPILATIONS PER MODULE	DT < AI = AT	0.088	= =	
MAXIMUM UNIQUE COMPILATIONS FOR ANY ONE MODULE	DT < AI = AT	0.118	DT < AI = AT	0.003

*"Confirmatory" aspects (presupposed to be quantitative indicators of important qualitative characteristics).
**Paired equal signs indicate the null outcome, AI=AT=DT.

Assuming each computer system operation involves a certain expenditure of the programmer's time and effort (e.g., effective terminal contact, test result evaluation), these aspects indirectly reflect human costs of development (at least that portion not devoted to design work).

The strength of the evidence supporting a difference with respect to location comparisons within this class is based on both (1) the near unanimity [8 out of 9 aspects] of the DT < AI = AT outcome and (2) the very low critical levels [< 0.025 for 5 aspects] involved. Indeed, the single exception among the location comparisons (outcome AI = AT = DT on the Identical Module Compilations aspect) is readily explained as a direct consequence of the fact that all teams made essentially similar use (or nonuse, in this case, since identical compilations were not uncommon) of the on-line storage capability (for saving relocatable modules and thus avoiding identical recompilations). This was expected since all teams had been provided with the same storage capability, but without any training or urging to use it. The conclusions on location comparisons within this class are interpreted as demonstrating that employment of the disciplined methodology by a programming team reduces the average costs, both machine and human, of software development, relative to both individual programmers and programming teams not employing the methodology. Examination of the raw data scores themselves indicates the magnitude of this reduction to be on the order of 2 to 1 (i.e., 50 percent) or better.

With respect to dispersion comparisons within this class, the evidence generally failed to make any distintction among the groups [AI = AT = DT on 7 out of 9 aspects]. These null conclusions in dispersion comparisons are interpreted as demonstrating that variability of software development costs, especially machine costs, is relatively insensitive to the factors of programming team size and degree of methodological discipline. The two exceptions on individual process aspects both deserve mention. The Miscellaneous Job Steps aspect showed an AT = DT < AI dispersion distinction among the groups, reflecting the wider-spread behavior (as expected) of individual programmers relative to programming teams in the area of building on-line tools to indirectly support software development (e.g., stand-alone module drivers, one-shot auxiliary computations, table generators, unanticipated debugging stubs). The Maximum Unique Compilations for Any One Module aspect showed a DT < AI = AT dispersion distinction among the groups at an extremely low critical level [<0.005], reflecting the lower variation (increased predictability) of the disciplined teams relative to the ad hoc teams and individuals in term of "worst case" compilation costs for any one module.

Class II Errors (program changes). Within Class II, the process aspect Program Changes, there is strong evidence of an important difference among the groups, again in favor of the disciplined methodology, with respect to average number of errors encountered during implementation. (See Table 2. For an explanation of how program changes are counted, see (9) in Appendix 2.) This aspect directly reflects the amount of textual revision to source code during (post-design) system development. Claiming that textual revisions are generally necessitated by errors encountered while building, testing, and debugging software, independent research[21] has demonstrated a high (rank order) correlation of total program changes (as counted automatically according to a specific algorithm) with total error occurrences (as tabulated manually from exhaustive scrutiny of source code and test results) during software implementation. This aspect is thus a reasonable measure of the relative number of programming errors encountered outside of design work. Assuming each textual revision involves a certain expenditure of the programmer's effort (e.g., planning the revision, on-line editing of source code), this aspect indirectly reflects the level of human effort devoted to implementation.

With respect to location comparison, the strength of the evidence supporting a difference among the groups is based on the very low critical level [< 0.005] for the DT < AI = AT outcome. The interpretation is that the disciplined methodology effectively reduced the average number of errors encountered during software implementation. This was expected since the methodology purposely emphasizes the criticality of the design phase and subjects the software design (code) to thorough reading and review prior to coding (key-in or testing), enhancing error detection and correction prior to implementation (testing).

With respect to dispersion comparison, no distinction among the groups was apparent, with the interpretation that variability in the number of errors encountered during implementation was essentially uniform across all three programming environments.

Class III: Gross Size. Within Class III, product aspects dealing with the gross size of the software at various hierarchical levels, there is evidence of certain consistent differences among the groups with

Table 2.
Conclusions for Class II: Errors (program changes).

PROGRAMMING ASPECT	LOCATION COMPARISON OUTCOME**	LOCATION CRITICAL LEVEL	DISPERSION COMPARISON OUTCOME**	DISPERSION CRITICAL LEVEL
* PROGRAM CHANGES	DT < AI = AT	0.003	=	=

*Confirmatory aspects.
**Paired equal signs indicate the null outcome.

respect to both average size and variability of size (see Table 3). As a class, these aspects directly reflect the number of objects and the average number of component (sub)objects per object, according to the hierarchical organization (imposed by the programming language) of the software itself into objects such as modules, routines, data variables, lines, statements, and tokens.

With respect to location comparisons within this class, the non-null conclusions [7 out of 17 aspects] are nearly unanimous [5 out of 7] in the AI < AT = DT outcome. The interpretation is that individuals tend to produce "smaller" software on the average than that produced by teams. It is unclear whether such spareness of expression, primarily in number of routines, global variables, and formal parameters, is advantageous or not. The two non-null exceptions to this AI < AT = DT trend deserve mention. The AT = DT < AI outcome on the Average Statements per Routine aspect is a simple consequence of the outcome for the number of Statements [AI = AT = DT] and the outcome for the number of Routines [AI < AT = DT], and it still fits the overall pattern of AI ≠ AT = DT on location differences on size aspects. On the Lines (of delivered source code) aspect, however, the DT = AI < AT distinction breaks the pattern. Since the number of statements was roughly the same for all three groups, this difference must be due mainly to the stylistic manner of arranging the source code (which was free-format with respect to line boundaries), to the amount of documentation comments within the source code, and to the number of lines taken up in data variable declarations.

With respect to dispersion comparisons within this class, the few aspects which do indicate any distinction among the group [5 out of 17 aspects] seem to concur on the AI = AT < DT outcome. This pattern, which associates increased variation in certain size aspects with the disciplined methodology, is somewhat surprising and lacks an intuitive explanation in terms of the experimental factors. The exception DT = AI < AT on Average Routine Per Module is really an exaggeration due to the fact that several teams in group AT built monolithic single-module systems, yielding rather inflated raw scores for this aspect. The exception AT < DT = AI on Statements is only a very slight trend, reflecting the fact that the AT products rather consistently contained the largest numbers of statements.

One overall observation for Class III is that while certain distinctions did consistently appear (especially for location but also for dispersion comparisons) at the middle levels of the hierarchical scale (routines, data variables, lines, and statements), no distinctions appeared at either the highest (modules) or lowest (tokens) levels of size. The null conclusions for size in modules and average module size seem attributable to the fact that particular programming applications often have standard approaches at the topmost conceptual levels which strongly influence the organization of software systems at this highest level of gross size. In the case at hand, a two-pass symbol-table/scanning/parsing/code-generation approach is extremely common for compilers, regardless of the particular parsing technique or symbol table organization employed, and the modules of nearly every system in the study distinctly reflected this common approach. The null conclusions for size in tokens is interpretable in view of Halstead's software science concepts,[22] according to which the program length N is predictable from the number η_2^* of basic input-output parameters and the language level λ. Since the functional specification, the application area, and the implementation language were all fixed in the study, both η_2^* and λ are essentially constant for each of the software systems, implying essentially constant

Table 3.
Conclusions for Class III: Gross Size.

PROGRAMMING ASPECT	LOCATION COMPARISON OUTCOME**	LOCATION CRITICAL LEVEL	DISPERSION COMPARISON OUTCOME**	DISPERSION CRITICAL LEVEL
* MODULES	= =		= =	
AVERAGE ROUTINES PER MODULE	= =		DT = AI < AT	0.021
AVERAGE GLOBAL VARIABLES PER MODULE	= =		= =	
* ROUTINES	AI < AT = DT	0.063	= =	
* AVERAGE STATEMENTS PER ROUTINE	AT = DT < AI	0.170	= =	
AVERAGE NONGLOBAL VARIABLES PER ROUTINE	= =		= =	
PARAMETER	AI < AT = DT	0.174	= =	
LOCAL	= =		= =	
DATA VARIABLES	AI < AT = DT	0.069	= =	
* GLOBAL	AI < AT = DT	0.147	AI = AT < DT	0.124
NONGLOBAL	= =		= =	
* PARAMETER	AI < AT = DT	0.127	AI = AT < DT	0.106
* LOCAL	= =		= =	
* LINES	DT = AI < AT	0.061	= =	
* STATEMENTS	= =		AT < DT = AI	0.195
* AVERAGE TOKENS PER STATEMENT	= =		AI = AT < DT	0.106
* TOKENS	= =		= =	

*Confirmatory aspects.
**Paired equal signs indicate the null outcome.

lengths N as measured in terms of operators and operands. Considering the number of tokens as roughly equivalent to program length N, the study's data seem to support the software science concepts in this instance.

Class IV: Control-Construct Structure. Within Class IV, product aspects dealing with the software's organization according to statements, constructs, and control structures, there are only a few distinctions between the groups (see Table 4).

With respect to location comparisons, the few [5 out of 24] aspects that showed any distinction at all were nearly unanimous [4 out of 5] in concluding DT = AI < AT. The one exception is the AT = DT < AI outcome for the Average Statement List Nesting Level aspect, indicating a slight tendancy for the ad hoc individuals to write code with greater statement nesting depth than either group of teams, apparently as a consequence of coding fewer but longer routines.

Essentially three particular issues are involved with the predominant DT = AI < AT outcome. First, the If Statement Count, If Statement Percentage, and Decisions aspects are all related to the frequency of programmer-coded decisions in the software product. Their common outcome DT = AI < AT is interpreted as demonstrating an important area in which the disciplined methodology causes a programming team to behave as an individual programmer. The number of decisions has been commonly accepted, and even formally derived,[23] as a measure of program complexity since more decisions create more paths through the code. Thus, the disciplined teams effectively reduced this form of software complexity, relative to the ad hoc teams. Second, the Return Statement Count aspect also distinguished between the ad hoc teams and the other two groups. Since the EXIT and RETURN statements are restricted forms of GOTOs, it seems to hint at another area in which the disciplined methodology improves conceptual control over program structure. Third, the Intrinsic (Proc) Call Statement Count aspect indicated a slight difference in the frequency of input-output operations coded, which seems interpretable only as a result of stylistic preferences.

With respect to dispersion comparisons, only two particular issues are involved. The Return Statement Count and Return Statement Percentage aspects both indicated a strong DT = AI < AT difference, suggesting that the frequency of these restricted GOTOs is an area in which the disciplined methodology reduces variability, causing a programming team to behave more like an individual programmer. The (Proc) Call Statement Count and Nonintrinsic (Proc) Call Statement Count aspects both showed a DT < AI = AT distinction among the groups, which is dealt with more appropriately within Class VIII below.

In summary of Class IV, the interpretation is that the functional component of control-conduct organi-

Table 4.
Conclusions for Class IV: Control-Construct Structure.

PROGRAMMING ASPECT	LOCATION COMPARISON OUTCOME**	LOCATION CRITICAL LEVEL	DISPERSION COMPARISON OUTCOME**	DISPERSION CRITICAL LEVEL
* STATEMENTS	= =		AT < DT = AI	0.195
STATEMENTS TYPE COUNTS:				
:=	= =		= =	
* IF	DT = AI < AT	0.078	= =	
* CASE	= =		= =	
* WHILE	= =		= =	
* EXIT	= =		= =	
(PROC) CALL	= =		DT < AT = AT	0.032
NONINTRINSIC	= =		DT < AI = AT	0.186
INTRINSIC	DT = AI < AT	0.173	= =	
* RETURN	DT = AI < AT	0.086	DT = AI < AT	0.003
STATEMENT TYPE PERCENTAGES:				
:=	= =		= =	
* IF	DT = AI < AT	0.106	= =	
* CASE	= =		= =	
* WHILE	= =		= =	
* EXIT	= =		= =	
(PROC) CALL	= =		= =	
NONINTRINSIC	= =		= =	
INTRINSIC	= =		= =	
* RETURN	= =		DT = AI < AT	0.040
* AVERAGE STATEMENT LIST NESTING LEVEL	AT = DT < AI	0.193	= =	
* DECISIONS	DT = AI < AT	0.146	= =	
FUNCTION CALLS	= =		= =	
NONINTRINSIC	= =		= =	
INTRINSIC	= =		= =	

*Confirmatory aspects.
**Paired equal signs indicate the null outcome.

zation is largely unaffected by the team size and methodological discipline factors, probably due to the overriding effect of project uniformity. However two facets of the control component that were influenced were the frequency of decisions (especially IF statements) and the frequency of restricted GOTOs (especially RETURN statements). For these aspects, the disiplined methodology altered the control structure (and reduced the complexity) of a team's product to that of an individual's product.

Class V: Data Variable Organization. Within Class V, product aspects dealing with data variables and their organization within the software, there are several distinctions among the groups, with an overall trend for both the location and dispersion comparisons (see Table 5). Data variable organization was, however, not emphasized in the disciplined methodology, nor in the academic course which the participants in group DT were taking.

With respect to location comparisons, all aspects showing any distinction at all were unanimous in concluding $AI \neq AT = DT$. The trend for individuals to differ from teams, regardless of the disciplined methodology, appears not only for the total number of data variables declared, but also for data variables at each scope level (global, parameter, local) in one fashion or another. In particular, the individuals seemed to program with fewer global variables and formal parameters than either group of teams. The difference regarding formal parameters is especially prominent, since it shows up for their raw count frequency, their normalized percentage frequency, and their average frequency per natural enclosure (routine).

With respect to dispersion comparisons, the apparent overall trend for aspects which show a distinction is toward the $AI = AT < DT$ outcome. No particular interpretation in view of the experimental factors seems appropriate. Exceptions to this trend ap-

Table 5.
Conclusions for Class V: Data Variable Organization.

PROGRAMMING ASPECT	LOCATION COMPARISON OUTCOME**	LOCATION CRITICAL LEVEL	DISPERSION COMPARISON OUTCOME**	DISPERSION CRITICAL LEVEL
DATA VARIABLES	AI < AT = DT	0.069	= =	
DATA VARIABLE SCOPE COUNTS:				
* GLOBAL	AI < AT = DT	0.147	AI = AT < DT	0.124
ENTRY	= =		= =	
MODIFIED	= =		= =	
UNMODIFIED	= =		= =	
NONENTRY	= =		= =	
MODIFIED	= =		= =	
UNMODIFIED	= =		= =	
MODIFIED	AI < AT = DT	0.161	= =	
UNMODIFIED	= =		= =	
NONGLOBAL	= =		= =	
* PARAMETER	AI < AT = DT	0.127	AI = AT < DT	0.106
VALUE	= =		= =	
REFERENCE	= =		AI < AT = DT	0.019
* LOCAL	= =		= =	
DATA VARIABLE SCOPE PERCENTAGES:				
* GLOBAL	= =		AI = AT < DT	0.075
ENTRY	= =		= =	
MODIFIED	= =		= =	
UNMODIFIED	= =		= =	
NONENTRY	= =		= =	
MODIFIED	= =		DT < AI = AT	0.021
UNMODIFIED	= =		= =	
MODIFIED	= =		= =	
UNMODIFIED	= =		= =	
NONGLOBAL	= =		AI = AT < DT	0.075
* PARAMETER	AI < AT = DT	0.150	AI = AT < DT	0.055
VALUE	= =		AI = AT < DT	0.094
REFERENCE	= =		AI < AT = DT	0.152
* LOCAL	AT = DT < AI	0.109	= =	
AVERAGE GLOBAL VARIABLES PER MODULE	= =		= =	
ENTRY	= =		= =	
NONENTRY	= =		= =	
MODIFIED	= =		DT = AI < AT	0.110
UNMODIFIED	= =		= =	
AVERAGE NONGLOBAL VARIABLES PER ROUTINE	= =		= =	
PARAMETER	AI < AT = DT	0.174	= =	
LOCAL	= =		= =	

*Confirmatory aspects. **Paired equal signs indicate the null outcome.

peared for both the raw count and percentage of call-by-reference parameters (both AI < AT = DT), as well as two other aspects.

Class VI: Packaging Structure. Within Class VI, product aspects dealing with modularity in terms of the packaging structure, there are essentially no distinctions among the groups, except for two location comparison issues (see Table 6). Most of the aspects in this class are also members of Class III, Gross Size, but are reconsidered here to focus attention upon the packaging characteristics of modularity (e.g., how the source code is divided into modules and routines, and what type of routines). The disciplined methodology did not explicitly include (nor did group DT's course work cover) concepts of modularization or criteria for evaluating good modularity; hence, no particular distinctions among the groups were expected in this area (Classes VI and VII).

With respect to location comparisons, the AI < AT = DT outcome for the Routines aspects, along with the companion outcome AT = DT < AI for the Average Statements per Routine aspect (as explained under Class III above), indicates one area of packaging that is apparently sensitive to the team size factor. Individual programmers built the system with fewer, but larger, routines (on the average) than either the ad hoc teams or the disciplined teams. The AI < AT = DT outcome for the Average Parameters per Routine aspect indicates that average "calling sequence" length, curiously enough, is another area of packaging sensitive to team size. With respect to dispersion comparisons, there really were no differences, since the single non-null outcome for Average Routines per Module is actually a fluke (raw scores for AT are exaggerated by the several monolithic systems) as explained above. The overall interpretation for this class is that modularity, in the sense of packaging code into routines and modules, is essentially unaffected by team size or methodological discipline, except for a tendency by individual programmers toward fewer, longer routines than programming teams.

Class VII: Invocation Organization. Within Class VII, product aspects dealing with modularity in terms of the invocation structure, there are two distinct trends for location comparisons, but no clear pattern for the dispersion comparison conclusions (see Table 7). This class consists of raw counts and average-per-routine frequencies for invocations (procedure call statements or function references in expressions) and is considered separately from the previous class since modularity involves not only the manner in which the system is packaged, but also the frequency with which the pieces are invoked. For the raw count frequencies of calls to intrinsic procedures and intrinsic routines, the trend is for the individuals and disciplined teams to exhibit fewer calls than the ad hoc teams. These intrinsic procedures are almost exclusively the input-output operations of the language, while the intrinsic functions are mainly data type conversion routines. The second trend for location comparisons occurs for two aspects related to the average frequency of calls to programmer-defined routines, in which the individuals display higher average frequency than either type of team. This seems coupled with group AI's preference for fewer but larger routines, as noted above. With respect to dispersion comparisons, several distinctions appear within this class, but no overall interpretation is readily apparent (except for a consistent DT <AI difference, with AT falling in between, leaning to one side or the other).

Class VIII: Inter-Routine Communication via Parameters. Within Class VIII, product aspects dealing with inter-routine communication via formal parameters, there are only a few distinctions among the groups (see Table 8). With respect to location comparisons, the total frequency of parameters and the average frequency of parameters per routine both show a difference. The interpretation is that the individual programmers tend to incorporate less inter-

Table 6.
Conclusions for Class VI: Packaging Structure.

PROGRAMMING ASPECT	LOCATION COMPARISON OUTCOME**	LOCATION CRITICAL LEVEL	DISPERSION COMPARISON OUTCOME**	DISPERSION CRITICAL LEVEL
* MODULES	= =		= =	
AVERAGE ROUTINE PER MODULE	= =		DT = AI < AT	0.021
AVERAGE GLOBAL VARIABLES PER MODULE	= =		= =	
* ROUTINES	AI < AT = DT	0.063	= =	
FUNCTIONS	= =		= =	
PROCEDURES	= =		= =	
FUNCTION PERCENTAGE	= =		= =	
PROCEDURE PERCENTAGE	= =		= =	
* AVERAGE STATEMENTS PER ROUTINE	AT = DT < AI	0.170	= =	
AVERAGE NONGLOBAL VARIABLES PER ROUTINE	= =		= =	
PARAMETER	AI < AT = DT	0.174	= =	
LOCAL	= =		= =	

*Confirmatory aspects.
**Paired equal signs indicate the null outcome.

routine communication via parameters, on the average, than either the ad hoc or the disciplined programming teams. With respect to dispersion comparisons, in addition to the difference in the raw count of parameters referred to in Class V, there is a strong difference in the variability of the number of call-by-reference parameters, also apparent in the percentages-by-type-of-parameter aspects. The interpretation is that the individual programmers were more consistent as a group in their use (in this case, avoidance) of reference parameters than either type of programming team.

Class IX: Inter-Routine Communication via Global Variables. Within Class IX, product aspects dealing with inter-routine communication via global variables, there are several differences among the groups, including two which indicate the beneficial influence of the disciplined methodology (see Table 9). This class is composed of aspects dealing with (1) frequency of globals, (2) average frequency of globals per module, (3) routine-global usage pairs (frequency of access paths from routines to globals), and (4) routine-global-routine data bindings[24] (frequency of logical bindings between two different routines via a global variable which is modified by the first routine and referenced by the second). These last two measures may be viewed as quantitative indicators of the general "globality" of global variables in a program, that is, the degree to which globals are accessible to exactly the routines that actually use them.

With respect to location comparisons, there is the AI < AT = DT distinction in sheer numbers of globals, particularly those modified during execution. However, when averaged per module, there appears to be no distinction in the normalized frequency of globals. The AI < AT = DT difference in the number of possible routine-global access paths makes sense as the result of group AI having both fewer routines and fewer globals. All three groups had essentially similar average levels of actual routine-global access paths, but several differences

Table 7.
Conclusions for Class VII: Invocation Organization.

PROGRAMMING ASPECT	LOCATION COMPARISON OUTCOME**	LOCATION CRITICAL LEVEL	DISPERSION COMPARISON OUTCOME**	DISPERSION CRITICAL LEVEL
INVOCATIONS	= =		AT = DT < AI	0.020
FUNCTION	= =		= =	
NONINTRINSIC	= =		= =	
INTRINSIC	= =		= =	
PROCEDURE	= =		DT < AI = AT	0.032
NONINTRINSIC	= =		DT < AI = AT	0.186
INTRINSIC	DT = AI < AT	0.173	= =	
NONINTRINSIC	= =		AT = DT < AI	0.051
INTRINSIC	DT = AI < AT	0.043	= =	
AVERAGE INVOCATIONS PER (CALLING) ROUTINE	= =		= =	
FUNCTION	= =		= =	
NONINTRINSIC	= =		= =	
INTRINSIC	= =		= =	
PROCEDURE	= =		= =	
NONINTRINSIC	= =		= =	
INTRINSIC	= =		DT < AI = AT	0.065
NONINTRINSIC	AT = DT < AI	0.169	= =	
INTRINSIC	= =		= =	
AVERAGE INVOCATIONS PER (CALLED) ROUTINE	AT = DT < AI	0.169	= =	
FUNCTION	AT = DT < AI	0.193	AT < DT = AI	0.141
PROCEDURE	= =		= =	

Table 8.
Conclusions for Class VIII: Communication via Parameters.

PROGRAMMING ASPECT	LOCATION COMPARISON OUTCOME**	LOCATION CRITICAL LEVEL	DISPERSION COMPARISON OUTCOME**	DISPERSION CRITICAL LEVEL
* PARAMETERS	AI < AT = DT	0.127	AI = AT < DT	0.106
VALUE	= =		= =	
REFERENCE	= =		AI < AT = DT	0.019
AVERAGE PARAMETERS PER ROUTINE	AI < AT = DT	0.174	= =	
PARAMETER PASSAGE TYPE PERCENTAGES:				
VALUE	= =		AI < AT = DT	0.160
REFERENCE	= =		AI < AT = DT	0.160

*Confirmatory aspects.
**Paired equal signs indicate the null outcome.

appear in the relative percentage (actual-to-possible ratio) category. These three instances of AT < DT = AI differences indicate that the degree of "globality" for global variables was higher for the individuals and the disciplined teams than for the ad hoc teams. Finally, another AT ≠ DT = AI difference appears for the frequency of possible routine-global-routine data bindings, indicating that the disciplined methodology effectively counteracted the ad hoc teams' increase in possible data coupling among routines. It may be noted that these last two categories of aspects, routine-global usage pair relative percentages and routine-global-routine data bindings, also reflect the quality of modularization, since good modularity should promote a higher degree of "globality" for globals and minimize the data coupling among routines. The interpretation here is that certain aspects of inter-routine communication via globals seem to be positively influenced, on the average, by the disciplined methodology.

With respect to dispersion comparisons, there is a diversity of differences in this class, without any unifying interpretation in terms of the experimental factors.

Table 9.
Conclusions for Class IX: Communication via Global Variables.

PROGRAMMING ASPECT	LOCATION COMPARISON OUTCOME**	LOCATION CRITICAL LEVEL	DISPERSION COMPARISON OUTCOME**	DISPERSION CRITICAL LEVEL
GLOBAL VARIABLES	AI < AT = DT	0.147	AI = AT < DT	0.124
ENTRY	= =		= =	
MODIFIED	= =		= =	
UNMODIFIED	= =		= =	
NONENTRY	= =		= =	
MODIFIED	= =		= =	
UNMODIFIED	= =		= =	
MODIFIED	AI < AT = DT	0.161	= =	
UNMODIFIED	= =		= =	
AVERAGE GLOBAL VARIABLES PER MODULE	= =		= =	
ENTRY	= =		= =	
NONENTRY	= =		= =	
MODIFIED	= =		DT = AI < AT	0.110
UNMODIFIED	= =		= =	
(ROUTINE, GLOBAL) ACTUAL USAGE PAIRS	= =		= =	
ENTRY	= =		= =	
MODIFIED	= =		= =	
UNMODIFIED	= =		= =	
NONENTRY	= =		= =	
MODIFIED	= =		= =	
UNMODIFIED	= =		= =	
MODIFIED	= =		AT < DT = AI	0.106
UNMODIFIED	= =		= =	
(ROUTINE, GLOBAL) POSSIBLE USAGE PAIRS	AI < AT = DT	0.122	AI < AT = DT	0.020
ENTRY	= =		= =	
MODIFIED	= =		= =	
UNMODIFIED	= =		= =	
NONENTRY	= =		AI < AT = DT	0.078
MODIFIED	= =		DT = AI < AT	0.051
UNMODIFIED	= =		AI < AT = DT	0.116
MODIFIED	= =		= =	
UNMODIFIED	= =		= =	
* (ROUTINE, GLOBAL) USAGE PAIR RELATIVE PERCENTAGE	= =		= =	
ENTRY	AT < DT = AI	0.082	= =	
MODIFIED	AT < DT < AI	0.123	= =	
UNMODIFIED	= =		= =	
NONENTRY	= =		= =	
MODIFIED	= =		= =	
UNMODIFIED	AT < DT = AI	0.154	= =	
MODIFIED	= =		= =	
UNMODIFIED	= =		= =	
(ROUTINE, GLOBAL, ROUTINE) DATA BINDINGS:				
* ACTUAL	= =		= =	
SUBFUNCTIONAL	= =		= =	
INDEPENDENT	= =		AI < AT = DT	0.196
* POSSIBLE	DT = AI < AT	0.186	DT = AI < AT	0.152
* RELATIVE PERCENTAGE	= =		= =	

*Confirmatory aspects.
**Paired equal signs indicate the null outcome.

Conclusion

Research into the effects of human factors on computer software is dependent upon adequate measurement of various "high-level" software properties, such as reliability, maintainability, modifiability, cost-effectiveness, complexity, comprehensibility, and readability. Because they are ill-defined, intangible, and multifaceted, it is very difficult to characterize and quantify these high-level properties directly.

There do exist numerous "low-level" programming aspects (such as those considered in this paper) which are so well-defined, tangible, and simple that they can readily be characterized and quantified on an individual and independent basis. Although each low-level aspect bears an intuitive relationship to one or more high-level properties, no single aspect (even the more promising ones) can adequately characterize and quantify a high-level property.

However, the coherent collective behavior of several low-level aspects, all related to the same high-level property, can serve as a credible (albeit indirect) indicator of the behavior of that high-level property. Thus it should be possible to conduct human factors research using statistically significant differences observed on several related low-level programming aspects to infer the existence of a differential effect upon the corresponding high-level software property.

The goal of this paper has been to demonstrate the feasibility of the above-outlined human factors research scenario, using the experimental results from a more extensive research project which quantitatively investigated different software development approaches. The study's findings reveal several programming aspects for which statistically significant differences do exist among the development approaches, and valuable insights have been gleaned regarding the effects of team programming and methodological discipline upon software reliability, complexity, etc.

Many of the low-level programming aspects, especially the "confirmatory" ones, demonstrate important characteristics for which the larger programming team size and the use of a disciplined methodology had beneficial effects on the development process and the developed product. The disciplined teams required fewer computer runs and apparently made fewer errors during software development than either the individual programmers or the ad hoc teams. The individual programmers and the disciplined teams both produced software with essentially the same number of decision statements, but software produced by the ad hoc teams contained greater numbers of decision statements. In fact, for no aspect was it concluded that the disciplined methodology impaired the effectiveness of a programming team or diminished the quality of the software product.

Based upon collective interpretation of classes of these low-level programming aspects, the study's findings indicate that the disciplined methodology increased software reliability beyond that achieved by either individual programmers or programming

teams using an ad hoc approach. As expected, ad hoc programming teams produced software having greater control flow complexity than individual programmers, but the disciplined methodology seemed effective in reducing this attendant software complexity.

Further research is now being undertaken, utilizing the study's data bank, to examine more elaborate software metrics, especially McCabe's measures of complexity[23] and Halstead's software science metrics.[22] Examination of the behavior of software product measures over the development time period is another direction open to human factors research. In addition, replications of the experiment itself are planned in order to strengthen and confirm this study's findings. ■

Acknowledgments

This work was supported in part by the Air Force Office of Scientific Research grant AFOSR-77-3181A to the University of Maryland. Computer time was supported in part through the facilities of the Computer Science Center of the University of Maryland.

The authors are grateful to the guest editor, Dr. John D. Gannon, for his patient assistance in emending the shortcomings of earlier drafts of this paper.

Appendix 1.
Measured programming aspects with statistical description of raw scores.

The following table lists the particular "low-level" programming aspects examined in the experiment, grouped by definitionally related categories. The parenthesized numbers refer to the explanatory notes in Appendix 2. The asterisks mark "confirmatory" aspects; "exploratory" aspects are unmarked.

Across-all-groups and within-each-group sample mean values and standard deviations supply a statistical description of the raw scores obtained in the experiment for each programming aspect.

PROGRAMMING ASPECTS	MEAN VALUES				STANDARD DEVIATIONS			
	ALL	AI	AT	DT	ALL	AI	AT	DT
DEVELOPMENT PROCESS ASPECTS								
(1) * COMPUTER JOB STEPS	157.0	185.5	223.5	75.6	93.8	90.7	84.2	25.5
(2) * MODULE COMPILATION	92.5	102.2	136.5	46.6	52.1	47.3	43.9	13.2
(3) * UNIQUE	73.1	81.3	108.3	35.9	41.4	32.5	37.9	11.6
(3) IDENTICAL	19.4	20.8	28.2	10.7	17.9	22.8	19.7	6.1
(4) * PROGRAM EXECUTION	60.3	76.0	82.3	28.0	44.6	43.8	51.5	14.5
(5) MISCELLANEOUS	4.2	7.3	4.7	1.0	5.5	7.6	4.9	1.5
(6) * ESSENTIAL	133.4	157.3	190.7	63.9	81.4	71.7	81.4	24.2
(3) AVERAGE UNIQUE COMPILATIONS PER MODULE	31.74	28.97	59.80	10.07	40.69	17.31	63.37	5.29
(3) MAXIMUM UNIQUE COMPILATIONS FOR ANY ONE MODULE	51.0	56.7	81.5	20.0	44.5	33.6	58.0	9.9
(9) * PROGRAM CHANGES	335.2	353.0	522.7	159.1	237.9	145.0	304.8	56.2
FINAL PRODUCT ASPECTS								
(10) * MODULES	4.5	3.8	5.3	4.3	3.7	3.1	5.6	2.1
(11) * ROUTINES	40.1	30.7	47.7	41.7	12.1	9.5	13.4	7.9
(12) ROUTINE TYPE COUNTS:								
(11) FUNCTION	6.2	4.7	8.8	5.3	5.2	5.7	5.1	4.8
(11) PROCEDURE	33.9	26.0	38.8	36.4	13.2	13.0	15.0	10.0
(12) ROUTINE TYPE PERCENTAGES:								
(11) FUNCTION	17.14	19.05	19.68	13.33	18.22	28.43	12.83	12.57
(11) PROCEDURE	82.86	80.95	80.32	86.67	18.22	28.43	12.83	12.57
AVERAGE ROUTINES PER MODULE	14.62	10.53	21.30	12.39	11.40	4.62	17.29	7.46
(14) * LINES	1323.5	1026.7	1676.5	1275.3	409.6	330.8	399.6	252.2
(15) * STATEMENTS	609.6	563.3	674.2	593.9	116.0	136.7	70.7	118.3
(16) STATEMENT TYPE COUNTS:								
:=	205.6	202.3	204.3	209.4	63.3	53.4	44.6	89.1
* IF	78.7	68.2	102.8	67.0	28.9	33.5	18.0	21.1
* CASE	6.6	7.0	5.2	7.4	3.6	3.7	3.0	4.1
* WHILE	25.2	25.2	27.2	23.6	5.6	6.7	5.2	5.3

	PROGRAMMING ASPECTS	MEAN VALUES				STANDARD DEVIATIONS			
		ALL	AI	AT	DT	ALL	AI	AT	DT
*	EXIT	2.7	4.7	3.2	0.7	4.6	5.8	5.4	1.3
(44)	(PROC) CALL	230.0	205.5	249.3	234.4	56.5	81.3	59.0	12.4
(23) (44)	NONINTRINSIC	187.0	167.3	192.3	199.3	54.2	74.8	64.2	10.5
(23) (44)	INTRINSIC	43.0	38.2	57.0	35.1	18.8	19.3	17.9	13.9
*	RETURN	60.7	50.3	82.2	51.3	22.6	9.0	30.0	6.5
(16)	STATEMENT TYPE PERCENTAGES:								
	:=	33.78	36.72	30.50	34.09	8.03	8.81	7.17	8.16
*	IF	12.56	11.55	15.22	11.16	3.20	3.62	2.15	2.37
*	CASE	1.10	1.32	0.77	1.20	0.55	0.68	0.41	0.46
*	WHILE	4.19	4.48	4.05	4.06	0.75	0.64	0.74	0.88
*	EXIT	0.40	0.70	0.42	0.13	0.61	0.78	0.68	0.22
	(PROC) CALL	38.00	36.02	36.97	40.59	7.48	7.97	8.08	6.91
(23)	NONINTRINSIC	31.09	29.58	28.35	34.74	8.41	9.60	8.27	7.27
(23)	INTRINSIC	6.91	6.45	8.62	5.84	2.56	2.59	2.88	1.65
*	RETURN	9.99	9.23	12.07	8.86	2.80	1.90	3.80	1.45
*	AVERAGE STATEMENTS PER ROUTINE	15.97	19.03	14.70	14.43	3.89	4.62	2.79	2.71
(26) *	AVERAGE STATEMENT LIST NESTING LEVEL	2.596	2.760	2.573	2.476	0.268	0.222	0.336	0.189
(27) *	DECISIONS	110.5	100.3	135.2	98.0	31.5	36.7	18.5	26.0
(44)	FUNCTION CALLS	90.8	90.2	104.8	79.4	44.1	65.2	33.8	31.6
(23) (44)	NONINTRINSIC	65.8	72.8	76.5	50.7	40.6	59.1	38.4	19.2
(23) (44)	INTRINSIC	25.0	17.3	28.3	28.7	13.5	9.0	15.3	14.1
(28) *	TOKENS	3340.2	3072.8	3707.0	3255.0	797.4	812.2	477.4	976.4
(28) *	AVERAGE TOKENS PER STATEMENT	5.45	5.45	5.50	5.40	0.55	0.37	0.43	0.80
(29)	INVOCATIONS	320.8	295.7	354.2	313.9	54.1	73.6	44.4	28.5
(11) (44)	FUNCTION	90.8	90.2	104.8	79.4	44.1	65.2	33.8	31.6
(23) (44)	NONINTRINSIC	65.8	72.8	76.5	50.7	40.6	59.1	38.4	19.2
(23) (44)	INTRINSIC	25.0	17.3	28.3	28.7	13.5	9.0	15.3	14.1
(11) (44)	PROCEDURE	230.0	205.5	249.3	234.4	56.5	81.3	59.0	12.4
(23) (44)	NONINTRINSIC	187.0	167.3	192.3	199.3	54.2	74.8	64.2	10.5
(23) (44)	INTRINSIC	43.0	38.2	57.0	35.1	18.8	19.3	17.9	13.9
(23)	NONINTRINSIC	252.8	240.2	268.8	250.0	43.0	61.3	43.7	19.7
(23)	INTRINSIC	68.0	55.5	85.3	63.9	21.0	18.3	17.7	17.4
(29)	AVERAGE INVOCATIONS PER (CALLING) ROUTINE	8.51	10.18	7.70	7.76	2.54	3.61	1.46	1.63
(11)	FUNCTION	2.57	3.53	2.32	1.97	2.07	3.48	0.89	0.88
(23)	NONINTRINSIC	1.92	2.90	1.72	1.26	1.89	3.12	1.01	0.52
(23)	INTRINSIC	0.66	0.63	0.60	0.73	0.34	0.41	0.22	0.40
(11)	PROCEDURE	5.93	6.67	5.38	5.76	1.38	1.59	1.44	0.99
(23)	NONINTRINSIC	4.81	5.38	4.07	4.96	1.27	1.51	1.08	1.04
(23)	INTRINSIC	1.13	1.30	1.32	0.83	0.56	0.74	0.58	0.23
(23) (44)	NONINTRINSIC	6.72	8.25	5.82	6.17	2.04	2.77	0.98	1.33
(23)	INTRINSIC	1.79	1.93	1.90	1.57	0.69	0.96	0.67	0.43
(29) (44)	AVERAGE INVOCATIONS PER (CALLED) ROUTINE	6.71	8.25	5.82	6.16	2.05	2.77	0.98	1.35
(11)	FUNCTION	16.86	23.02	9.70	17.71	11.14	10.87	3.36	13.21
(11)	PROCEDURE	6.33	8.07	5.12	5.87	3.21	5.10	1.31	1.73
(32)	DATA VARIABLES	94.6	68.2	103.7	109.6	33.3	27.4	19.2	36.7
(37)	DATA VARIABLE SCOPE COUNTS:								
(33) *	GLOBAL	35.7	24.7	35.3	45.6	20.0	5.6	7.3	29.9
(33)	ENTRY	9.2	7.0	9.8	10.4	10.0	9.0	14.0	8.0
(35)	MODIFIED	8.5	6.3	8.7	10.3	9.0	7.9	11.9	8.0
(35)	UNMODIFIED	0.6	0.7	1.2	0.1	1.7	1.0	2.9	0.4
(33)	NONENTRY	26.6	17.7	25.5	35.1	20.2	6.7	15.4	28.6
(35)	MODIFIED	18.1	15.0	18.7	20.1	9.2	6.7	10.8	10.1
(35)	UNMODIFIED	8.5	2.7	6.8	15.0	16.4	1.8	9.3	25.5
(35)	MODIFIED	26.6	21.3	27.3	30.4	9.5	3.7	4.7	14.0
(35)	UNMODIFIED	9.2	3.3	8.0	15.1	16.2	2.3	8.7	25.4
(33)	NONGLOBAL	58.9	43.5	68.3	64.0	24.5	23.5	14.9	28.2
(33) *	PARAMETER	25.1	13.0	29.3	31.9	17.0	11.1	10.8	21.1
(36)	VALUE	17.7	11.8	21.3	19.7	11.5	10.9	9.4	13.0
(36)	REFERENCE	7.4	1.2	8.0	12.1	12.1	1.6	9.9	17.0

	PROGRAMMING ASPECTS	MEAN VALUES				STANDARD DEVIATIONS			
		ALL	AI	AT	DT	ALL	AI	AT	DT
(33) *	LOCAL	33.8	30.5	39.0	32.1	10.8	13.3	8.0	10.5
(37)	DATA VARIABLE SCOPE PERCENTAGES:								
(33) *	GLOBAL	38.05	38.82	34.33	40.59	12.95	9.93	5.85	19.25
(33)	ENTRY	9.86	8.28	10.50	10.67	11.42	9.90	16.35	9.13
(35)	MODIFIED	9.10	7.58	9.02	10.47	9.97	8.80	13.19	9.17
(35)	UNMODIFIED	0.77	0.73	1.48	0.20	2.11	1.16	3.63	0.53
(33)	NONENTRY	28.17	30.48	23.83	29.91	16.08	16.67	13.09	19.33
(35)	MODIFIED	20.64	26.32	18.12	17.94	10.90	15.40	11.01	3.31
(35)	UNMODIFIED	7.54	4.17	5.72	11.99	13.15	3.49	7.28	20.66
(35)	MODIFIED	29.74	33.92	27.13	28.40	8.33	8.37	6.52	9.38
(35)	UNMODIFIED	8.31	4.88	7.20	12.19	12.94	3.67	6.78	20.55
(33)	NONGLOBAL	61.95	61.18	65.67	59.41	12.95	9.93	5.85	19.25
(33) *	PARAMETER	24.50	16.67	27.67	28.50	12.19	7.54	7.45	16.19
(36)	VALUE	17.81	15.22	20.03	18.13	8.18	7.65	5.95	10.52
(36)	REFERENCE	6.69	1.47	7.62	10.37	11.14	2.13	9.31	15.89
(33) *	LOCAL	37.44	44.53	37.98	30.89	9.45	7.05	6.66	9.41
	AVERAGE GLOBAL VARIABLES PER MODULE	12.62	8.52	16.98	12.40	10.76	3.83	16.31	8.80
(33)	ENTRY	1.68	1.42	1.00	2.49	1.65	1.53	1.10	1.98
(33)	NONENTRY	10.97	7.12	16.02	9.94	11.34	4.67	17.16	8.97
(35)	MODIFIED	9.13	7.48	11.97	8.10	5.78	3.25	9.10	3.20
(35)	UNMODIFIED	3.51	1.03	5.03	4.31	7.31	0.86	9.84	8.36
	AVERAGE NONGLOBAL VARIABLES PER ROUTINE	14.74	14.38	14.70	15.07	4.93	6.72	3.10	5.19
(33)	PARAMETER	5.90	3.97	6.23	7.27	3.27	2.26	2.20	4.22
(33)	LOCAL	8.84	10.40	8.48	7.80	3.39	5.14	2.10	2.22
(39)	PARAMETER PASSAGE TYPE PERCENTAGES:								
(36)	VALUE	81.12	91.23	76.73	76.21	25.21	14.49	25.33	32.36
(36)	REFERENCE	18.88	8.77	23.27	23.79	25.21	14.49	25.33	32.36
(40)	(ROUTINE, GLOBAL) ACTUAL USAGE PAIRS	142.5	120.7	151.3	153.7	36.1	35.4	19.8	42.7
(33)	ENTRY	55.3	48.3	53.3	63.0	53.7	53.6	74.4	39.3
(35)	MODIFIED	54.8	48.3	52.3	62.4	52.6	51.5	72.9	39.4
(35)	UNMODIFIED	0.8	0.8	1.0	0.6	1.7	1.3	2.4	1.5
(33)	NONENTRY	86.9	71.5	98.0	90.7	57.2	47.2	66.4	62.6
(35)	MODIFIED	75.4	67.0	90.2	69.9	54.1	44.8	61.8	60.0
(35)	UNMODIFIED	11.6	4.5	7.8	20.9	26.6	4.2	10.1	43.1
(35)	MODIFIED	130.2	115.3	142.5	132.3	32.2	32.6	17.5	40.0
(35)	UNMODIFIED	12.4	5.3	8.8	21.4	26.3	4.2	9.4	42.8
(40)	(ROUTINE, GLOBAL) POSSIBLE USAGE PAIRS	828.1	488.3	1195.3	804.4	518.2	182.9	680.4	375.5
(33)	ENTRY	269.4	170.7	368.3	269.3	394.0	234.1	646.7	227.8
(35)	MODIFIED	262.6	163.5	355.5	267.9	390.6	219.7	645.0	228.8
(35)	UNMODIFIED	6.8	7.2	12.8	1.4	19.4	16.1	31.4	3.8
(33)	NONENTRY	558.6	317.7	827.0	535.1	535.6	225.6	779.5	425.4
(35)	MODIFIED	397.7	277.8	563.3	358.6	362.9	182.5	474.2	369.3
(35)	UNMODIFIED	160.9	39.8	263.7	176.6	302.8	50.4	434.8	299.8
(35)	MODIFIED	660.3	441.3	918.8	626.4	379.3	139.9	472.9	335.8
(35)	UNMODIFIED	167.7	47.0	276.5	178.0	299.7	50.5	426.5	298.9
(40) *	(ROUTINE, GLOBAL) USAGE PAIR RELATIVE PERCENTAGE	22.07	26.38	16.80	22.90	9.59	7.49	9.50	10.32
(33)	ENTRY	28.97	43.23	13.83	29.71	18.99	17.55	13.39	15.57
(35)	MODIFIED	29.26	43.35	14.57	29.77	19.03	17.32	14.57	15.64
(35)	UNMODIFIED	6.42	12.35	1.30	5.71	17.27	26.75	3.18	15.12
(33)	NONENTRY	20.12	23.63	15.98	20.64	8.59	7.36	7.50	9.98
(35)	MODIFIED	23.02	24.57	21.78	22.76	8.76	8.53	9.43	9.57
(35)	UNMODIFIED	12.94	23.85	3.48	11.70	18.84	29.64	3.90	10.46
(35)	MODIFIED	23.89	27.25	19.22	25.03	9.12	6.98	9.72	9.79
(35)	UNMODIFIED	13.25	22.67	4.78	12.44	18.03	27.95	3.81	12.28
(41)	(ROUTINE, GLOBAL, ROUTINE) DATA BINDINGS:								
(42) *	ACTUAL	419.9	344.7	426.7	478.6	357.2	142.9	333.5	513.2
(43)	SUBFUNCTIONAL	262.6	249.0	270.0	268.0	208.0	107.4	217.8	283.4
(43)	INDEPENDENT	157.3	95.7	156.7	210.6	163.9	39.0	124.7	243.4
(42) *	POSSIBLE	21310	11014	42391	12065	26463	6477	39828	9691
(42) *	RELATIVE PERCENTAGE	3.66	3.83	1.92	5.00	3.55	2.31	1.49	5.14

Appendix 2.
Explanatory notes for measured programming aspects

The following numbered paragraphs, keyed to the list of aspects in Appendix 1, describe the programming aspects examined in the experiment, with definitions for the nontrivial or unfamiliar ones. Various system- or language-dependent terms (e.g., module, routine, intrinsic, entry) are also defined here. (The experiment was part of a larger software research project, and aspects not relevant to this report have been omitted. The numbering scheme used is consistent with that of the total report, published elsewhere.[12])

(1) A *computer job step* is a single indivisible activity performed on a computer at the operating system command level; it is inherent to the development effort and involves a nontrivial expenditure of computer or human resources. Only module compilations and program executions were counted.

(2) A *module compilation* is an invocation of the implementation language processor on the source code of an individual module. Only compilations of modules comprising the final software product (or logical predecessors thereof) were counted.

(3) All module compilations are categorized as either *identical* or *unique* depending on whether or not the source code compiled is textually identical to that of a previous compilation.

(4) A *program execution* is an invocation of a complete programmer-developed program (after the necessary compilation(s) and collection or link-editing) upon some test data.

(5) A *miscellaneous job step* is an auxiliary compilation or execution of something other than the final software product.

(6) An *essential job step* is a computer job step that involves the final software product (or logical predecessors thereof) and could not have been avoided (by off-line computation or by on-line storage of previous compilations or results).

(9) The *program changes* metric[21] is defined in terms of textual revisions made to the source code of a module during the development period, from the time that module is first presented to the computer system, to the completion of the project. The rules for counting program changes are such that one program change should represent approximately one conceptual change to the program. Each of the following is counted as a single program change: modification of a single statement, insertion of contiguous statements, or modification of a single statement followed immediately by insertion of contiguous statements. However, the following are not counted as program changes: deletion of contiguous statements, insertion of standard output statements or special compiler-provided debugging directives, insertion of blank lines or comments, modification of comments, and reformatting of statements without semantic alteration. Program changes are counted automatically according to a specific algorithm which symbolically compares the source code from each pair of consecutive compilations of a particular module (or logical predecessor thereof).

(10) A *module* is a separately compiled portion of the complete software system. In the implementation language SIMPL-T, a typical module is a collection of the declarations of several global variables and the definitions of several routines.

(11) A *routine* is a collection of source code statements, together with declarations for the formal parameters and local variables manipulated by those statements, that may be invoked as an operational unit. In the implementation language SIMPL-T, a routine is either a value-returning *function* (invoked via reference in an expression) or else a non-value-returning *procedure* (invoked via the CALL statement); recursive routines are allowed and fully supported.

(12) The group of aspects named Routine Type Counts, etc., gives the absolute number of programmer-defined routines of each type. The group of aspects named Routine Type Percentages, etc., gives the relative percentage of each type of routine, compared with the total number of programmer-defined routines.

(14) The Lines aspect counts every textual line of delivered source code in the final product, including comments, compiler directives, variable declarations, executable statements, etc.

(15) The Statements aspect counts only the executable constructs in the source code of the complete final product. These are high-level, structured-programming statements—including simple statements, such as assignment and procedure call, as well as compound statements, such as IFTHENELSE and WHILEDO, which have other statements nested within them. The implementation language SIMPL-T allows exactly seven different statement types (referred to by their distinguishing keyword or symbol) covering assignment (:=), alternation-selection (IF, CASE), iteration (WHILE, EXIT), and procedure invocation (CALL, RETURN). Input-output operations are accomplished via calls to certain intrinsic procedures.

(16) The group of aspects named Statement Type Counts, etc., gives the absolute number of executable statements of each type. The group of aspects named Statement Type Percentages, etc., gives the relative percentage of each type of statement, compared with the total number of executable statements.

(23) *Intrinsic* means provided and defined by the implementation language; *nonintrinsic* means provided and defined by the programmer. Nearly all of the intrinsic procedures in the implementation language SIMPL-T perform input-output operations and external data file manipulations. All of the intrinsic functions in SIMPL-T perform data type coercions and character string operations.

(26) In the implementation language SIMPL-T, both simple (e.g., assignment) and compound (e.g., IFTHENELSE) statements may be nested inside other compound statements. A particular *nesting level* is associated with each statement list, starting at 1 for the statement list at the outermost level of each routine and increasing by 1 for successively nested statement lists.

(27) The Decisions aspect counts the numbers of IF, CASE, and WHILE statements within the complete source code.

(28) *Tokens* are the basic syntactic entities—such as keywords, operators, parentheses, identifiers, etc.—that occur in a program statement.

(29) An *invocation* is simply the syntactic occurrence of a construct by which either a programmer-defined or built-in routine is invoked from within another routine; both procedure calls and function references are counted. Invocations are (sub)categorized by the type of routine being invoked.

(32) A *data variable* is an individually named scalar or array of scalars. The Data Variables aspect counts each data variable declared in the final software product once, regardless of its type, structure, or scope. Note that each array is counted as a single data variable.

(33) In the implementation language SIMPL-T, data variables can have any one of essentially four levels of *scope*—entry global, nonentry global, parameter, and local—depending on where and how they are declared in the program. *Global* variables are accessible by name to each of the routines in the module in which they are declared. *Entry globals* are actually accessible by name to each of the routines in several (two or more) modules: the module which declares it ENTRY, plus each of the modules which declare it EXTernal. *Nonentry globals* are accessible by name only within the module in which they are delcared. *Nonglobal* variables are accessible by name only to the single routine in which they are declared: formal *parameters* have values that are somehow related to a calling routine (by the parameter passing mechanism), while *locals* have values that are completely isolated from any other routine.

(35) *Modified* means referred to, at least once in the program source code, in such a manner that the value of the data variable would be (re)set when (and if) the appropriate statements were to be executed. *Unmodified* means referred to, throughout the program source code, in such a manner that the value of the data variable could never be (re)set during execution. In the implementation language SIMPL-T, unmodified globals must be initialized (within their declarations) in order to be useful as "named constants."

(36) The implementation language SIMPL-T allows two types of parameter passage. Pass-by-*value* means that the value of the actual argument is simply copied (upon invocation) into the corresponding formal parameter (which thereafter behaves like a local variable for all intents and purposes), with the effect that the called routine cannot modify the value of the calling routine's actual argument. Pass-by-*reference* means that the address of the actual argument—which must be a variable rather than an expression—is passed (upon invocation) to the called routine, with the effect that any changes made by the called routine to the corresponding formal parameter will be reflected in the value of the calling routine's actual argument (upon return). In SIMPL-T, formal parameters that are scalars are normally (default) passed by value, but they may be explicitly declared to be passed by reference; formal parameters which are arrays are always passed by reference.

(37) The group of aspects named Data Variable Scope Counts, etc., gives the absolute number of declared data variables according to each level of scope. The group of aspects named Data Variable Scope Percentages, etc., gives the relative percentage of variables at each scope level, compared with the total number of declared variables.

(39) The group of aspects named Parameter Passage Type Percentages, etc., gives the percentages of each type of parameter relative to the total number of parameters declared in the program.

(40) A routine-global *usage pair* (r,g) is an instance of a global variable g being used by a programmer-defined routine r (i.e., the global is either modified (set) or accessed (fetched) at least once within the statements of the routine). Each usage pair represents a unique "use connection" between a global and a routine.

In this study, routine-global usage pairs were (sub)categorized by the type of global involved and were counted in three different ways. First, the (Routine, Global) Actual Usage Pairs aspects count the absolute numbers of true usage pairs (r,g) : the global variable g is actually used by routine r. Second, the (Routine, Global) Possible Usage Pairs aspects count the absolute numbers of potential usage pairs (r,g), given the program's global variables and their declared scope: the scope of global variable g merely contains routine r, so that r could potentially modify or access g. These counts of possible usage pairs are computed as the sum of the number of routines in each global's scope. Third, the (Routine, Global) Usage Pair Relative Percentage aspects are a way of normalizing the number of usage pairs since these measures are simply the ratios (expressed as percentages) of actual usage pairs to possible usage pairs.

(41) A routine-global-routine *data binding*[24] $(r1,g,r2)$ is an occurrence of the following arrangement in a program: a programmer-defined routine $r1$ modifies (sets) a global variable g that is also accessed (fetched) by a programmer-defined routine $r2$, with $r1$ different from $r2$. The binding $(r1,g,r2)$ is different from the binding $(r2,g,r1)$ which may also exist; occurrences such as (r,g,r) are not counted as data bindings.

(42) In this study, routine-global-routine data bindings were counted in three different ways. First, the Actual count is the absolute number of true data bindings $(r1,g,r2)$: the global variable g is actually modified by routine $r1$ and actually accessed by routine $r2$. Second, the Possible count is the absolute number of potential data bindings $(r1,g,r2)$, given the program's global variables and their declared scope: the scope of global variable g merely contains both routine $r1$ and routine $r2$, so that $r1$ could potentially modify g and $r2$ could potentially access g. This count of Possible data bindings is computed as the sum of terms $s*(s-1)$ for each global, where s is the number of routines in that global's scope; thus, it is fairly sensitive (numerically speaking) to the total number of routines in a program. Third, the Relative Percentage is a way of normalizing the number of data bindings since it is simply the quotient (expressed as a percentage) of the actual data bindings divided by the possible data bindings.

(43) Actual data bindings are (sub)categorized depending on the invocation relationship between the two routines. A data binding $(r1,g,r2)$ is *subfunctional* if either of the two routines $r1$ or $r2$ can invoke the other, whether directly or indirectly (via a chain of intermediate invocations involving other routines). In this situation, the functioning of the one routine may be viewed as contributing to the overall functioning of the other routine. A data binding $(r1,g,r2)$ is *independent* if neither of the two routines $r1$ or $r2$ can invoke the other, whether directly or indirectly. The transitive closure of the call graph among the routines of a program is employed to make this distinction between subfunctional and independent.

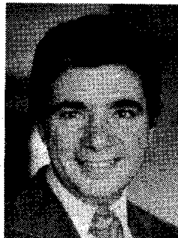
(44) Several instances of duplicate programming aspects exist in the tables presented in this paper. That is, certain logically unique aspects appear a second time with another name, in order to provide alternative views of the same metric and to achieve a certain degree of completeness within a set of related aspects. Listed below are the pairs of duplicate programming aspects that were considered in this study:

FUNCTION CALLS	<=>	INVOCATIONS, FUNCTION
NONINTRINSIC	<=>	NONINTRINSIC
INTRINSIC	<=>	INTRINSIC
STATEMENT TYPE COUNTS, (PROC) CALL	<=>	INVOCATIONS, PROCEDURE
NONINTRINSIC	<=>	NONINTRINSIC
INTRINSIC	<=>	INTRINSIC
AVERAGE INVOCATIONS PER (CALLING) ROUTINE, NONINTRINSIC	<=>	AVERAGE INVOCATIONS PER (CALLED) ROUTINE

By definition, the data scores obtained for any pair of duplicate aspects will be identical, and thus the same statistical conclusions will be reached for both aspects.

References

1. G. M. Weinberg, *The Psychology of Computer Programming,* Van Nostrand Reinhold, New York, 1971.
2. M. E. Sime, T. R. G. Green, and D. J. Guest, "Psychological Evaluation of Two Conditional Constructs Used in Computer Languages," *Int'l J. Man-Machine Studies,* Vol. 5, No. 1, Jan. 1973, pp. 105-113.
3. L. M. Weissman, "Psychological Complexity of Computer Programs: an Experimental Methodology," *ACM SIGPLAN Notices,* Vol. 9, No. 6, June 1974, pp. 25-36.
4. L. M. Weissman, "A Methodology for Studying the Psychological Complexity of Computer Programs," PhD thesis, CSRG-37, Dept. of Computer Science, University of Toronto, Aug. 1974.
5. J. D. Gannon and J. J. Horning, "Language Design for Programming Reliability," *IEEE Trans. Software Eng.,* Vol. 1, No. 2, June 1975, pp. 179-191.
6. B. Shneiderman, "Exploratory Experiments in Programmer Behavior," *Int'l J. Computer and Information Sciences,* Vol. 5, No. 2, June 1976, pp. 123-143.
7. J. D. Gannon, "An Experimental Evaluation of Data Type Conventions," *Comm. ACM,* Vol. 20, No. 8, Aug. 1977, pp. 584-595.
8. T. R. G. Green, "Conditional Program Statements and Their Comprehensibility to Professional Programmers," *J. Occupational Psychology,* Vol. 50, 1977, pp. 93-109.
9. S. B. Sheppard, B. Curtis, P. Milliman, and T. Love, "Modern Coding Practices and Programmer Performance," *Computer,* Vol. 12, No. 12, Dec. 1979, pp. 41-49.
10. B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems,* Winthrop Publishing Co., Cambridge, Mass., 1980.
11. S. S. Stevens, "On the Theory of Scales of Measurement," *Science,* Vol. 103, 1946, pp. 677-680.
12. V. R. Basili and R. W. Reiter, Jr., "Investigating Software Development Approaches," Technical Report TR-688, Dept. of Computer Science, University of Maryland, Aug. 1978.
13. R. W. Reiter, Jr., "Empirical Investigation of Computer Program Development Approaches and Computer Programming Metrics," PhD diss., Dept. of Computer Science, University of Maryland, Dec. 1979.
14. V. R. Basili and R. W. Reiter, Jr., "A Controlled Experiment in Quantitatively Comparing Software Development Approaches" (to be published in *IEEE Trans. Software Eng.,* 1980).
15. R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice,* Addison-Wesley, Reading, Mass. 1979.
16. V. R. Basili and F. T. Baker, *Tutorial of Structured Programming,* IEEE Catalog No. 75CH1049-6, Tutorial from COMPCON 75 Fall (revised 1977).
17. F. P. Brooks, Jr., *The Mythical Man-Month,* Addison-Wesley, Reading, Mass., 1975.
18. V. R. Basili and A. J. Turner, *SIMPL-T, A Structured Programming Language,* Paladin House Publishers, Geneva, Ill., 1976.
19. J. W. Tukey, "Analyzing Data: Sanctification or Detective Work?" *American Psychologist,* Vol. 24, No. 2, Feb. 1969, pp. 83-91.
20. H. D. Mills, "The Complexity of Programs," in *Program Test Methods,* edited by W.C. Hetzel, Prentice-Hall, Englewood Cliffs, N. J., 1973, pp. 225-238.
21. H. E. Dunsmore and J. D. Gannon, "Experimental Investigation of Programming Complexity," *Proc. ACM-NBS Sixteenth Annual Technical Symposium: Systems and Software,* Washington, DC, June 1977, pp. 117-125.
22. M. Halstead, *Elements of Software Science,* Elsevier Computer Science Library, 1977.
23. T. J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.,* Vol. 2, No. 4, Dec. 1976, pp. 308-320.
24. W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured Design," *IBM Systems J.,* Vol. 13, No. 2, 1974, pp. 115-139.



Victor R. Basili is an associate professor in the Department of Computer Science, University of Maryland, College Park. With the department since 1970, he has been involved in the design and development of several software projects, including the SIMPL family of structured programming languages, the graph algorithmic language GRAAL, and the SL/1 language for the CDC Star. He is currently involved in the measurement and evaluation of software development at NASA/Goddard Space Flight Center, and has acted as a consultant at the Institute for Computer Applications in Science and Engineering at NASA/Langley Research Center, the Naval Research Laboratories, the Naval Systems Weapons Laboratory, Computer Sciences Corporation, and IBM Federal Systems Division.

A member of ACM, the IEEE Computer Society, and the American Association of University Professors, Basili received the BS degree in mathematics from Fordham College, the MS degree in mathematics from Syracuse University, and the PhD degree in computer science from the University of Texas, Austin, in 1961, 1963, and 1970, respectively.

Robert W. Reiter, Jr., is a faculty research assistant with the Department of Computer Science at the University of Maryland. As exemplified in his dissertation, his current research interests cover empirical study in software engineering, software development and maintenance methodology, and software metrics. During the course of his graduate studies, he also contributed to the enhancement and maintenance of the SIMPL family of transportable extendable compilers and to the initial formulation of the Software Engineering Laboratory at NASA/Goddard Space Flight Center.

A member of ACM and the IEEE Computer Society, Reiter received an SB degree in mathematics from MIT in 1972 and the MS and PhD degrees in computer science from the University of Maryland in 1976 and 1979, respectively.