through 5.9. For each aspect class, it is interesting to jointly interpret the individual outcomes in an overall manner in order to see something of how these higher-level issues are affected by the factors of team size and methodological discipline.

Class I:

Within Class I (process aspects dealing with COMPUTER JOB STEPS), there is strong evidence of an important difference among the groups, in favor of the disciplined methodology, with respect to average development costs. 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. 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 (a) the near unanimity [8 out of 9 aspects] of the DT < AI = AT outcome and (b) the very low critical levels [<.025 for 5 aspects] involved. Indeed, the single exception among the location comparisons (AI = AT = DT on COMPUTER JOB STEPS\MODULE COMPILATIONS\IDENTICAL) is readily explained as a direct consequence of the fact that all teams made essentially similar usage (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 identical 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%) or better.

With respect to dispersion comparisons within this class, the evidence generally failed to make any distinctions 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 COMPUTER JOB STEPS\MISCELLANEOUS aspect showed a 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, etc.). The MAX UNIQUE COMPILATIONS F.A.O. MODULE aspect showed a DT < AI = AT dispersion distinction among the groups at an extremely low critical level [<.005], reflecting the lower variation (increased predictability) of the disciplined teams relative to the ad hoc teams and individuals in terms of "worst case" compilation costs for any one module. The additional AI < AT distinction for this comparison is clearly attributable to the fact that several teams in group AT build monolithic single-module systems, yielding rather inflated raw scores for this aspect.

Class II:

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. Appendix 1 contains a detailed explanation of how program changes are counted. This aspect directly reflects the amount of textual revision to the source code during (postdesign) development. Claiming that textual revisions are generally necessitated by errors encountered while building, testing, and debugging software, recent research [Dunsmore and Gannon 77] has confirmed 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 [<.005] for the DT < AI = AT outcome. The additional trend toward AI < AT is much less pronounced in the data. 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 through 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 considered.

Class III:

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 respect to both average size and variability of size. 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, segments, 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 software which is smaller (in certain ways) on the average than that produced by teams. It is unclear whether such spareness of expression, primarily in segments, global variables, and formal parameters, is advantageous or not. The two non-null exceptions to this AI < AT = DT trend deserve mention, since the one is only nominally exceptional and actually supportive of the tendency upon closer inspection, while the other indicates a size aspect in which the disciplined methodology enabled programming teams to break out of the pattern of distinction from individual programmers. The AT = DT < AI outcome on AVERAGE STATEMENTS PER SEGMENT is a simple consequence of the outcome for the number of STATEMENTS (AI = AT = DT) and the outcome for the number of SEGMENTS (AI < AT = DT) and it still fits the overall pattern of AI ≠ AT = DT on location differences on size aspects. On the LINES aspect, the DT = AI < AT distinction breaks the pattern since DT is associated with AI and

not with AT. 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 groups [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 SEGMENTS PER MODULE is really an exaggeration due to the fact of several AT teams implementing monolithic single-module systems, as mentioned above. 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 [segments, 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 tasks or application domains often have certain standard approaches at the topmost conceptual levels which strongly influence the organization of software systems at this highest level of gross size. In this case, the two-pass symbol-table/scanning/parsing/code-generation approach is extremely common for language translation problems (i.e., compilers), regardless of the particular parsing technique or symbol table organization employed, and the modules of nearly every system in the study directly reflected this common approach. The null conclusions for size in tokens is interpretable in view

of Halstead's software science concepts [Halstead 77], 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 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:

Within Class IV (product aspects dealing with the software's organization according to statements, constructs, and control structures), there are only a few distinctions made between the groups.

With respect to location comparisons, the few [5 out of 24] aspects that showed any distinction at all were unanimous in concluding DT = AI < AT. Essentially, three particular issues were involved. The STATEMENTS TYPE COUNTS\IF, STATEMENT TYPE PERCENTAGES\IF, 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 like an individual programmer. The number of decisions has been commonly accepted, and even formalized [McCabe 76], as a measure of program complexity since more decisions create more paths through the code. Thus, the disciplined methodology effectively reduced the average complexity from what it otherwise would have been. The STATEMENT TYPE COUNTS\RETURN aspect indicates a difference between the ad hoc teams and the other two groups. Since the EXIT and RETURN statements are restricted forms of GOTOs, this difference seems to hint at another area in which the disciplined methodology improves conceptual control over program structure. The STATEMENT TYPE

COUNTS\(PROC)CALL\INTRINSIC aspect also indicates a slight trend in the area of the frequency of input-output operations, which seems interpretable only as a result of stylistic differences.

With respect to dispersion comparisons, only two particular issues were involved. The STATEMENT TYPE COUNTS\RETURN and STATEMENT TYPE PERCENTAGE\RETURN 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 STATEMENT TYPE COUNTS\(PROC)CALL and STATEMENT TYPE COUNTS\(PROC)CALL\NONINTRINSIC aspects both showed a DT < AI = AT distinction among the groups, which is dealt with more appropriately within Class VII below.

In summary of Class IV, the interpretation is that the functional component of control-construct organization is largely unaffected by the team size and methodological discipline factors, probably due to the overriding effect of project/task uniformity/commonality. 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 disicplined methodology altered the control structure (and reduced the complexity) of a team's product to that of an individual's product.

Class V:

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

Class VI:

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. Most of the aspects in this class are also members of Class III, Gross Size, but are (re)considered here to focus attention upon the packaging characteristics of modularity (i.e., how the source code is divided into modules and segments, what type of segments, etc.). 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 SEGMENTS aspects, along with the companion outcome AT = DT < AI for the AVERAGE STATEMENTS PER SEGMENT 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 (on the average), segments than either the ad hoc teams or the disciplined

teams. The AI < AT = DT outcome for the AVERAGE NONGLOBAL VARIABLES PER SEGMENT\PARAMETER 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 SEGMENTS 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 segments and modules, is essentially unaffected by team size or methodological discipline, except for a tendency by individual programmers toward fewer, longer segments than programming teams.

Class VII:

Within Class VII (product aspects dealing with modularity in terms of the invocation structure), there are two distinction trends for location comparisons, but no clear pattern for the dispersion comparison conclusions. This class consists of raw counts and average-per-segment 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 (a third aspect is actually redundant) 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 reflection of a DT < AI difference, with AT falling in between, leaning one side or the other).

Class VIII:

Within Class VIII (product aspects dealing with inter-segment communication via formal parameters), there are only a few distinctions among the groups. With respect to location comparisons, the total frequency of parameters and the average frequency of parameters per segment both show a difference. The interpretation is that

> the individual programmers tend to incorporate less inter-segment 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:

Within Class IX (product aspects dealing with inter-segment communication via global variables), there are several differences among the groups, including two which indicate the beneficial influence of the disciplined methodology. This class is composed of aspects dealing with (a) frequency of globals, (b) average frequency of globals per module, (c) segment-global usage pairs

(frequency of access paths from segments to globals), and (d) segment-global-segment data bindings [Stevens, Myers, and Constantine 74; pp. 118-119] (frequency of logical bindings between two different segments via a global variable which is modified by the first segment and referenced by the second).

With respect to location comparisons, there is the AI < AT = DT distinction in sheer numbers of globals, particularly globals which are modified during execution, as noted in Class V. However, when averaged per module, there appears to be no distinction in the frequency of globals. The AI < AT = DT difference in the number of possible segment-global access paths makes sense as the result of group AI having both fewer segments and fewer globals. All three groups had essentially similar average levels of actual segment-global access paths, but several differences 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 segment-global-segment data bindings, indicating a positive effect of the disciplined methodology in reducing the possible data coupling among segments. It may be noted that these last two categories of aspects, segment-global usage relative percentages and segment-global-segment data bindings, also reflect upon the quality of modularization, since good modularity should promote the degree of "globality" for globals and minimize the data coupling among segments. The interpretation here is that

> certain aspects of inter-segment communication via globals seems 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.

VI. Concluding Remarks

A practical methodology was designed and developed for experimentally and quantitatively investigating the software development phenomenon. It was employed to compare three particular software development environments and to evaluate the relative impact of a particular disciplined methodology (made up of so-called modern programming practices). The experiments were successful in measuring differences among programming environments and the results support the claim that disciplined methodology effectively improves both the process and product of software development.

One way to substantiate the claim for improved process is to measure the effectiveness of the particular programming methodology via the number of bugs initially in the system (i.e., in the initial source code) and the amount of effort required to remove them. (This criteria was independently suggested by Professor M. Shooman of Polytechnic Institute of New York while speaking recently on the subject of sofware reliability models.) Although neither of these measures was directly computed, they are each closely associated with one of the process aspects considered in the study: PROGRAM CHANGES and ESSENTIAL JOB STEPS, respectively. The statistical conclusions (on location comparison) for both these aspects affirmed DT < AI = AT outcomes at very low (<.01) significance levels, indicating that on the average the disciplined teams measured lower than either the ad hoc individuals or the ad hoc teams which both measured about the same. Thus, the evidence collected in this study strongly confirms the effectiveness of the disciplined methodology in building reliable software efficiently.

The second claim, that the product of a disciplined team should closely resemble that of a single individual since the disciplined methodology assures a semblence of conceptual integrity within a programming team, was partially substantiated.

In many product aspects the products developed using the disciplined methodology were either similar to or tended toward the products developed by the individuals. In no case did any of the measures show the disciplined teams' products to be worse than those developed by the ad hoc teams. It is felt that the superficiality of most of the product measures was chiefly responsible for the lack of stronger support for this second claim. The need for product measures with increased sensitivity to critical characteristics of software is very clear.

The results of these experiments will be used to guide further experiments and will act as a basis for analysis of software development products and processes in the Software Engineering Laboratory at NASA/GSFC [Basili et al. 77]. The intention is to persue this type of research, especially extending the study to include more sophisticated and promising programming aspects, such as Halstead's software science quantities [Halstead 77] and other software complexity metrics [McCabe 76].

References

[Baker 75] F.T. Baker. Structured Programming in a Production Programming Environment. IEEE Transactions on Software Engineering, Vol. 1, No. 2 (June 1975), pp. 241-252.

[Basili and Baker 75] V.R. Basili and F.T. Baker. Tutorial of Structured Programming. IEEE Catalog No. 75CH1049-6, Eleventh IEEE Computer Society Conference (COMPCON), 1975.

[Basili and Turner 75] V.R. Basili and A.J. Turner. Iterative Enhancement: A Practical Technique for Software Development. IEEE Transactions on Software Engineering, Vol. 1, No. 4 (December 1975), pp. 390-396.

[Basili and Turner 76] V.R. Basili and A.J. Turner. SIMPL-I, A Structured Programming Language. Paladin House Publishers, Geneva, Illinois. 1976.

[Basili et al. 77] V.R. Basili, M.V. Zelkowitz, F.E. McGarry, R.W. Reiter, Jr., W.F. Truszkowski, and D.L. Weiss. The Software Engineering Laboratory. Technical Report TR-535, Department of Computer Science, University of Maryland. May, 1977.

[Belady and Lehman 76] L.A. Belady and M.M. Lehman. A Model of Large Program Development. IBM Systems Journal, Vol. 15 (1976), No. 3, pp. 225-251.

[Brooks 75] F.P. Brooks, Jr. The Mythical Man-Month. Addison-Wesley Publishing Co., Reading, Massachusetts. 1975.

[Conover 71] W.J. Conover. Practical Nonparametric Statistics. John Wiley & Sons Inc., New York. 1971.

[Dahl, Dijkstra, and Hoare 72] O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare. Structured Programming. Academic Press, New York. 1972.

[Daley 77] E.B. Daley. Management of Software Development. IEEE Transactions on Software Engineering, Vol. 3, No. 3 (May 1977), pp. 229-242.

[Dunsmore and Gannon 77] H.E. Dunsmore and J.D. Gannon. Experimental Investigation of Programming Complexity. Proceedings of ACM-NBS Sixteenth Annual Technical Symposium: Systems and Software (June 1977), Washington, D.C., pp. 117-125.

[Halstead 77] M. Halstead. Elements of Software Science. Elsevier Computer Science Library. 1977.

[Jackson 75] M.A. Jackson. Principles of Program Design. Academic Press, New York. 1975.

[Kirk 68] R.E. Kirk. Experimental Design: Procedures for the Behavioral Sciences. Wadsworth Publishing Co., Belmont, California. 1968.

[Linger, Mills, and Witt 79] R.C. Linger, H.D. Mills, and B.I. Witt. Structured Programming Theory and Practice. Addison-Wesley (to be published). 1979.

[McCabe 76] T.J. McCabe. A Complexity Measure. IEEE Transactions on Software Engineering, Vol. 2, No. 4 (December 1976), pp. 308-320.

[Mills 73] H.D. Mills. The Complexity of Programs. in Program Test Methods, edited by W.C. Hetzel., pp. 225-238. Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1973.

[Myers 75] G.J. Myers. Reliable Software through Composite Design. Petrocelli/Charter. 1975.

[Myers 78] G.J. Myers. A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections. Communications of the ACM, Vol. 21, No. 9 (September 1978), pp. 760-768.

[Nemenyi et al. 77] P. Nemenyi, S.K. Dixon, N.B. White, Jr., and M.L. Hedstrom. Statistics from Scratch. Holden-Day, San Francisco, California. 1977.

[Ostle and Mensing 75] B. Ostle and R.W. Mensing. Statistics in Research, Third Edition. Iowa State University Press, Ames, Iowa. 1975.

[Sheppard et al. 78] S.B. Sheppard, M.A. Borst, B. Curtis, and T. Love. Factors Influencing the Understandability and Modifiability of Computer Programs. Human Factors (to be published). 1978.

[Shneiderman et al. 77] B. Shneiderman, R. Mayer, D. McKay, and P. Heller. Experimental Investigations of the Utility of Detailed Flowcharts in Programming. Communications of the ACM, Vol. 20, No. 6 (June 1977), pp. 373-381.

[Siegel 56] S. Siegel. Nonparametric Statistics: for the Behavioral Sciences. McGraw-Hill Book Co., New York. 1956.

[Stevens, Myers, and Constantine 74] W.P. Stevens, G.J. Myers, and L.L. Constantine. Structured Design. IBM Systems Journal, Vol. 13 (1974), No. 2, pp. 115-139.

[Tukey 69] J.W. Tukey. Analyzing Data: Sanctification or Detective Work? American Psychologist, Vol. 24, No. 2 (February 1969), pp. 83-91.

[Wirth 71] N. Wirth. Program Development by Stepwise Refinement. Communications of the ACM, Vol. 14, No. 4 (April 1971), pp. 221-227.

Appendix 1. Explanatory Notes for the Programming Aspects

The following numbered paragraphs, keyed to the list of aspects in Table 1, explain in detail the programming aspects considered in the study. Various system- or language-dependent terms (e.g., module, segment, intrinsic, entry) are also defined here.

(1) A computer job step is a single activity performed on a computer at the operating system command level which is inherent to the development effort and involves a nontrivial expenditure of computer or human resources. Typical job steps might include text editing, module compilation, program collection or link-editing, and program execution; however, operations such as querying the operating system for status information or requesting access to on-line files would not be considered as job steps. In this study, only module compilations and program executions are counted as COMPUTER JOB STEPS.

(2) A module compilation is an invocation of the implementation language processor on the source code of an individual module. In this study, only compilations of modules comprising the final software product (or logical predecessors thereof) are counted as COMPUTER JOB STEPS\MODULE COMPILATIONS.

(3) All MODULE COMPILATIONS are classified as either IDENTICAL or UNIQUE depending on whether or not the source code compiled is textually identical to that of a previous compilation. During the development process, each unique compilation was necessary in some sense, while an identical compilation could have been logically avoided by saving the relocatable output of a previous compilation for later reuse (except in the situation of undoing source code revisions after they have been tested and found to be erroneous or superfluous).

(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. Only job steps counted as COMPUTER JOB STEPS, but not counted as COMPUTER JOB STEPS\MODULE COMPILATIONS or COMPUTER JOB STEPS\ PROGRAM EXECUTIONS, are counted as COMPUTER JOB STEPS\ MISCELLANEOUS.

(6) An essential job step is a computer job step which 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). In this study, the number of ESSENTIAL JOB STEPS is the sum of the number of COMPUTER JOB STEPS\MODULE COMPILATIONS\UNIQUE plus the number of COMPUTER JOB STEPS\PROGRAM EXECUTIONS.

(7) The number of AVERAGE UNIQUE COMPILATIONS PER MODULE is simply the number of COMPUTER JOB STEPS\MODULE COMPILATIONS\UNIQUE divided by the number of MODULES.

(8) The number of MAX UNIQUE COMPILATIONS F.A.O. MODULE is simply the maximum number of unique compilations for any one module of the final software product. F.A.O. stands for "for any one". Each unique compilation is associated (either directly or as a logical predecessor) with a particular module of the final product; their sum is computed for each module; and the maximum of the sums is taken.

(9) The program changes metric [Dunsmore and Gannon 77] is defined in terms of textual revisions in 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 --which are reproduced below from the paper referenced above with the kind permission of the authors-- are such that one program change

should represent approximately one conceptual change to the program.

The following each represent a single program change:
(a) one or more changes to a single statement,
(A single statement in a program represents a single concept and even multiple character changes to that statement represent mental activity with a single concept.)
(b) one or more statements inserted between existing statements,
(The contiguous group of statements inserted probably corresponds to a single abstract instruction.)
(c) a change to a single statement followed by the insertion of new statements.
(This instance probably represents a discovery that an existing statement is insufficient and that it must be altered and supplemented in order to achieve the single concept for which it was produced.)

However, the following are not counted as program changes:
(a) the deletion of one or more existing statements,
(Statements which are deleted must usually be replaced with other statements elsewhere. The inserted statements are counted; counting deletions as well would give double weight to such a change. Occasionally statements are deleted but not replaced; these are probably being used for debugging purposes and their deletion takes no great mental activity.)
(b) the insertion of standard output statements or special compiler-provided debugging directives,
(These are occasionally inserted in a wholesale fasion during debugging. When the problem is discerned, these are then all removed, and the actual ststement change takes place.)
(c) the insertion of blank lines, insertion of comments, revision of comments, and reformatting without alteration of existing statements.
(These are all judged to be cosmetic in nature.)

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). Thus the total number of program changes is a measure of the amount of textual revision to source code during (postdesign) system development.

(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 segments. [In this study, only those modules which comprise the final product are counted as MODULES.]

(11) A segment is a collection of source code statements, together with declarations for the formal parameters and local variables manipulated by those statements, which may be invoked as

an operational unit. In the implementation language SIMPL-T, a segment is either a value-returning function (invoked via reference in an expression) or else a non-value-returning procedure (invoked via the CALL statement), and recursive segments are allowed and fully supported. The segment, function, and procedure of SIMPL-T correspond to the (sub)program, function, and subroutine of FORTRAN, respectively.

(12) The group of aspects named SEGMENT TYPE COUNTS, etc., gives the absolute number of programmer-defined segments of each type. The group of aspects named SEGMENT TYPE PERCENTAGES, etc., gives the relative percentage of each type of segment, compared with the total number of programmer-defined segments. The second group of aspects is computed from the first by simply dividing by the number of SEGMENTS, as a way of normalizing the segment type counts.

(13) Since segment definitions in the implementation language SIMPL-T occur within the context of a module, this provids a natural way to normalize (or average) the raw counts of segments. The AVERAGE SEGMENTS PER MODULE aspect represents the number of segments in a typical module. It is computed in the obvious way.

(14) The number of LINES is the total count of every textual line in the source code of the complete final product, including comments, compiler directives, variable declarations, executable statements, etc.

(15) The number of STATEMENTS 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 if-then-else and while-do-- 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. The second group of aspects is computed from the first by simply dividing by the number of STATEMENTS, as a way of normalizing the statement type counts.

(17) As mentioned above, the := symbol denotes the assignment statement. It assigns the value of the expression on the right hand side to the variable on the left hand side.

(18) Both if-then and if-then-else constructs are counted as IF statements. Each IF statement allows the execution of either the then- or else-part statements, depending upon its Boolean expression.

(19) The CASE statement provides for selection from several alternatives, depending upon the value of an expression. In the implementation language SIMPL-T, exactly one of the alternatives (or an optional else-part) is selected per execution of a CASE, a list of constants is explicitly given for each alternative, and selection is based upon the equality of the expression value with one of the constants. A case construct with n alternatives is logically and semantically equivalent to a certain pattern of n nested if-then-else constructs.

(20) The WHILE statement is the only iteration or looping construct provided by the implementation language SIMPL-T. It allows the statements in the loop body to be executed repeatedly (zero or more times) depending upon a Boolean expression which is reevaluated at every iteration; the loop may also be terminated

via an EXIT statement. Each WHILE statement may be optionally labeled with a designator (referenced by EXIT statements) which uniquely identifies it from other nested WHILE statements.

(21) The EXIT statement allows the abnormal termination of iteration loops by unconditional transfer of control to the statement immediately following the WHILE statement. Thus it is a very restricted form of GOTO. This exiting may take place from any depth of nested loops, since the EXIT statement may optionally name a designator which identifies the loop to be exited; without such a designator only the immediately enclosing loop is exited.

(22) Since there are two types of segments in the implementation language SIMPL-T, there are two types of "calls" or segment invocations. Procedures are invoked via the CALL statement, and functions are invoked via reference in an expression. The counts for these separate constructs are reported separately as the (PROC)CALL and FUNCTION CALL aspects, and jointly as the INVOCATIONS aspect.

(23) Intrinsic means provided and defined by the implementation language; nonintrinsic means provided and defined by the programmer. These terms are used to distinguish built-in procedures or functions (which are supported by the compiler and utilized as primitives) from segments (which are written by the programmer himself). Nearly all of the intrinsic procedures provided by the implementation language SIMPL-T perform input-output operations and external data file manipulations. All of the intrinsic functions provided by SIMPL-T perform data type conversions and character string manipulations.

(24) The RETURN statement allows the abnormal termination of the current segment by unconditional resumption of the previously executing segment. Thus it is another very restricted form of GOTO. Within a function, a RETURN statement must specify an expression, the value of which becomes the value returned for the function invocation. Within a procedure, a RETURN statement must

not specify such an expression. Additionally, a simple RETURN statement is optional at the textual end of procedures; it will be implicitly assumed if not explicitly coded. In this study, the total number of explicitly coded and implicitly assumed RETURN statements, both from functions and procedures combined, is counted.

(25) The AVERAGE STATEMENTS PER SEGMENT aspect provides a way of normalizing the number of statements relative to their natural enclosure in a program, the segment. The measure also represents the length, in executable statements, of a typical segment of the program.

(26) In the implementation language SIMPL-T, both simple (e.g., assignment) and compound (e.g., if-then-else) statements may be nested inside other compound statements. A particular nesting level is associated with each statement, starting at 1 for a statement at the outermost level of each segment and increasing by 1 for successively nested statments. Nesting level can be displayed visually via proper and consistent indentation of the souce code listing.

(27) The number of DECISIONS is simply the sum of the numbers of IF, CASE, and WHILE statements within the complete source code. Each of these statements represents a unique (possibly repeated) run-time decision coded by the programmer. This count is closely associated with a recently proposed complexity metric [McCabe 76] which essentially reflects the number of binary-branching decisions represented in the source code.

(28) Tokens are the basic syntactic entities --such as keywords, operators, parentheses, identifiers, etc.-- that occur in a program statement. The average number of tokens per statement may be viewed as an indication of how much "information" a typical statement contains, how "powerful" a typical statement is, or how concisely the statements in general are coded.

(29) An invocation is simply the syntactic occurrence of a construct by which either a programmer-defined segment or a built-in routine is invoked from within another segment; both procedure calls and function references are counted as INVOCATIONS. They are (sub)classified by the type (i.e., function or procedure, nonintrinsic or intrinsic) of segment or routine being invoked.

(30) The group of aspects named AVG INVOCATIONS PER (CALLING) SEGMENT, etc., represents one way to normalize the absolute number of invocations. These aspects reflect the number of calls to programmer-defined segments and built-in routines from a typical programmer-defined segment. They are (sub)classified by the type of segment or routine being invoked. The measures for this group of aspects are computed by simply dividing each of the corresponding measures in the INVOCATIONS aspect group by the number of SEGMENTS.

(31) The group of aspects named AVG INVOCATIONS PER (CALLED) SEGMENT, etc., represents another way to normalize the absolute number of invocations. These aspects reflect the number of calls to a typical programmer-defined segment from other segments. They are (sub)classified by the type (i.e., function or procedure) of segment being invoked.

(32) A data variable is an individually named scalar or array of scalars. In the implementation language SIMPL-T, (a) there are three data types for scalars --integer, character, and (varying length) string--, (b) there is one kind of data structure (besides scalar) --single dimensional array, with zero-origin subscript range--, and (c) there are several levels of scope (as explained in note 33 below) for data variables. In addition, all data variables in a SIMPL-T program must be explicitly declared, with attributes fully specified. The number of DATA VARIABLES is computed by counting each of the data variables declared in the final software product once, regardless of 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. Note that the notion of scope deals only with static accessibility by name; the effective accessibility of any variable can always be extended by passing it as a parameter between segments. The scope levels are explained here (and presented in the aspect (sub)classifications) via a hierarchy of distinctions.

The primary distinction is between global and nonglobal. Global variables are accessible by name to each of the segments in the module in which they are declared. Nonglobal variables are accessible by name only to the single segment in which they are declared.

Global varaibles are secondarily distinguished into entry and nonentry. Entry globals are actually accessible by name to each of the segments in several (two or more) modules: the module which declared it ENTRY, plus each of the modules which declared it EXTernal (as explained in note 34 below). Nonentry globals are accessible by name only within the module in which they are declared.

Nonglobal variables are secondarily distinguished into formal parameter and local. Formal parameters are accessible by name only within the enclosing (called) segment, but their values are not completely unrelated to the calling segment (as explained in note 36 below). Locals are accessible by name only within the enclosing segment, and their values are completely isolated from any other segment.

(34) Entry means that the data variable [or segment] is declared to be accessible from within other separately compiled modules (in which it must be explicitly declared as EXTernal). Nonentry means that the data variable [or segment] is accessible only within the module in which it is declared [or defined]. In this study these terms are used pertaining only to global variables. "Entry global" actually constitutes an extra level of scope beyond "nonentry global". [Although the implementation

language SIMPL-T does allow the EXTernal attribute to be declared for local variables --just the enclosing segment has access to a global declared in a different module--, it is an extremely obscure and rarely used feature; it never occurred in any of the final software products examined in this study.]

(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. Data variables can be (re)set only by (a) being the "target" of an assignment statement, (b) being passed by reference to some programmer-defined segment or built-in routine, or (c) being named in an "input statement." This third case is really covered by the second case since all the "input statements" in SIMPL-T are actually calls to certain intrinsic procedures with passed-by-reference parameters. 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. These terms are used pertaining to global data variables; any global variable is allowed to have an initial value (constants only) specified in its declaration. Globals which are initialized but UNMODIFIED are particularly useful in SIMPL-T programs, serving as "named constants."

(36) The inplementation 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 segment'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 segment's actual argument (upon return). In SIMPL-T, formal parameters which 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. The second group of aspects is computed from the first by simply dividing by the number of DATA VARIABLES, as a way of normalizing the data variable scope counts.

(38) Since data variable declarations in the implementation language SIMPL-T may only appear in certain contexts within the program --globals in the context of a module and and nonglobals in the context of a segment--, this provides a natural way to normalize (or average) the raw counts of data variables. The group of aspects named AVERAGE GLOBAL VARIABLES PER MODULE, etc., represent the number of globals declared for a typical module. They are computed by simply dividing each of the corresponding raw counts of global data variables by the number of MODULES. The group of aspects named AVERAGE NONGLOBAL VARIABLES PER SEGMENT, etc., represent the number of nonglobals declared for a typical segment. They are computed by simply dividing each of the corresponding raw counts of nonglobal data variables by the number of SEGMENTS.

(39) Since there are two types of parameter passing mechanisms in the implementation language SIMPL-T (as explained in note 36 above), it is desirable to normalize their raw frequencies into relative percentages, indicating the programmer's degree of "preference" for one type or the other. 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. They are computed in the obvious way.

(40) A segment-global usage pair (p,r) is simply an instance of a global variable r being used by a segment p (i.e., the global is either modified (set) or accessed (fetched) at least once within the statements of the segment). Each usage pair represents a unique "use connection" between a global and a segment. Usage pairs are (sub)classified by the type (i.e., entry or nonentry, modified or unmodified) of global data variable involved.

In this study, segment-global usage pairs were counted in three different ways. First, the (SEG,GLOBAL) ACTUAL USAGE PAIR counts are the absolute numbers of true usage pairs (p,r): the global variable r is actually used by segment p. They represent the true frequencies of use connections within the program. Second, the (SEG,GLOBAL) POSSIBLE USAGE PAIR counts are the absolute numbers of potential usage pairs (p,r), given the program's global variables and their declared scope: the scope of global variable r simply contains segment p, so that segment p could potentially modify or access r. These counts of possible usage pairs are computed as the sum of the number of segments in each global's scope. They represent a sort of "worst case" frequencies of use connections. Third, the (SEG,GLOBAL) USAGE RELATIVE PERCENTAGE counts 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. They represent the frequencies of true use connections relative to potential use connections. These usage pair relative percentage metrics are empirical estimates of the likelihood that an arbitrary segment uses (i.e., sets or fetches the value of) an arbitrary global variable.

(41) A segment-global-segment data binding (p,r,q) is an occurrence of the following arrangement in a program [Stevens, Myers, and Constantine 74]: a segment p modifies (sets) a global variable r which is also accessed (fetched) by a segment q, with segment p different from segment q. The existence of a data binding (p,r,q) indicates that the behavior of segment q is probably dependent on the performance of segment p because of the data variable r, whose value is set by p and used by q. The

binding (p,r,q) is different from the binding (q,r,p) which may also exist; occurrences such as (p,r,p) are not counted as data bindings. Thus each (SEG,GLOBAL,SEG) DATA BINDING represents a unique communication path between a pair of segments via a global variable. The total number of (SEG,GLOBAL,SEG) DATA BINDINGS reflects the degree of a certain kind of "connectivity" (i.e., between segment pairs via globals) within a complete program.

(42) In this study, segment-global-segment data bindings were counted in three different ways. First, the ACTUAL count is the absolute number of true data bindings (p,r,q): the global variable r is actually modified by segment p and actually accessed by segment q. It represents the degree of true connectivity in the program. Second, the POSSIBLE count is the absolute number of potential data bindings (p,r,q), given the program's global variables and their declared scope: the scope of global variable r simply contains both segment p and segment q, so that segment p could potentially modify r and segment q could potentially access r. 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 segments in that global's scope; thus, it is fairly sensitive (numerically speaking) to the total number of SEGMENTS in a program. It represents a sort of "worst case" degree of potential connectivity. 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. It represents the degree of true connectivity relative to potential connectivity.

(43) Actual data bindings are (sub) classified as "subfunctional" or "independent" depending on the invocation relationship between the two segments. A data binding (p,r,q) is subfunctional if either of the two segments p or q can invoke the other, whether directly or indirectly (via a chain of intermediate invocations involving other segments). In this situation, the function of the one segment may be viewed as contributing to the overall function of the other segment. A data binding (p,r,q) is

independent if neither of the two segments p or q can invoke the other, whether directly or indirectly. The transitive closure of the call graph among the segments of a program is employed to make this distinction between subfunctional and independent.

(44) There exist several instances of duplicate programming aspects in the Table 1 listing. 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. For example, the FUNCTION CALLS aspect and the STATEMENT TYPE COUNTS\(PROC)CALL aspect are listed (and categorized appropriately) from the viewpoint of the various type of constructs which comprise the the implementation language. But the very same metrics can be considered from the unifying viewpoint of the various subtype frequencies for segment invocations, and thus it is desirable to include the duplicate aspects INVOCATIONS\FUNCTIONS and INVOCATIONS\PROCEDURES as part of the natural categorization of INVOCATIONS. Listed below are the pairs of duplicate programming aspects that were considered in this study:

1. FUNCTION CALLS
   <=> INVOCATIONS\FUNCTION
2. FUNCTION CALLS\NONINTRINSIC
   <=> INVOCATIONS\FUNCTION\NONINTRINSIC
3. FUNCTION CALLS\INTRINSIC
   <=> INVOCATIONS\FUNCTION\INTRINSIC
4. STATEMENT TYPE COUNTS\(PROC)CALL
   <=> INVOCATIONS\PROCEDURE
5. STATEMENT TYPE COUNTS\(PROC)CALL\NONINTRINSIC
   <=> INVOCATIONS\PROCEDURE\NONINTRINSIC
6. STATEMENT TYPE COUNTS\(PROC)CALL\INTRINSIC
   <=> INVOCATIONS\PROCEDURE\INTRINSIC
7. AVG INVOCATIONS PER (CALLING) SEGMENT\NONINTRINSIC
   <=> AVG INVOCATIONS PER (CALLED) SEGMENT

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

Appendix 2. English Statements for the Non-Null Conclusions

The following numbered sentences simply provide English translations for the non-null location comparisons presented in symbolic equation form in Table 2.1. They may be skimmed by the reader since they do not add to the information appearing in the table.

(1) According to the SEGMENTS aspect, the individuals (AI) organized their software into noticeably fewer routines (i.e., functions or procedures) than either the ad hoc teams (AT) or the disciplined teams (DT).

(2) Both the ad hoc teams (AT) and the disciplined teams (DT) declared a noticeably larger number of data variables (i.e., scalars or arrays of scalars) than the individuals (AI), according to the DATA VARIABLES aspect.

(3) In particular, a definite trend toward this same difference was apparent in the number of global variables, the number of global variables whose values could be modified during execution, and the number of formal parameter variables, according to the DATA VARIABLE SCOPE COUNTS\GLOBAL, DATA VARIABLE SCOPE COUNTS\GLOBAL\MODIFIED, and DATA VARIABLE SCOPE COUNTS\NONGLOBAL\PARAMETER aspects, respectively.

(4) A trend existed for the individuals (AI) to have a smaller percentage of formal parameters compared to the total number of declared data variables than either the ad hoc teams (AT) or the disciplined teams (DT), according to the DATA VARIABLE SCOPE PERCENTAGES\NONGLOBAL\PARAMETER aspect.

(5) According to the AVERAGE NONGLOBAL VARIABLES PER SEGMENT\ PARAMETER aspect, there was a trend for the individuals (AI) to have fewer formal parameters per routine than did either the ad hoc teams (AT) or the disciplined teams (DT).

(6) A definite trend existed for the individuals (AI) to have fewer possible segment-global usage pairs (i.e., potential access of a global variable by a routine) than either the ad hoc teams (AT) or the disciplined teams (DT), according to

the (SEG,GLOBAL) POSSIBLE USAGE PAIRS aspect.

(7) According to the AVERAGE STATEMENTS PER SEGMENT aspect, the individuals (AI) displayed a trend toward having a greater number of statements per routine than did either the ad hoc teams (AT) or the disciplined teams (DT).

(8) There existed slight trends toward more calls to programmer-defined routines per calling routine and per called routine for the individuals (AI) than for either the ad hoc teams (AT) or the disciplined teams (DT), according to the AVG INVOCATIONS PER (CALLING) SEGMENT\NONINTRINSIC and AVG INVOCATIONS PER (CALLED) SEGMENT aspects.

(9) In addition, a very slight trend existed for the individuals (AI) to have more calls to programmer-defined functions, averaged per programmer-defined function, than either the ad hoc teams (AT) or the disciplined teams (DT), according to the AVG INVOCATIONS PER (CALLED) SEGMENT\FUNCTION aspect.

(10) According to the DATA VARIABLE SCOPE PERCENTAGES\NONGLOBAL\ LOCAL aspect, the individuals (AI) had a larger percentage of local variables compared to the total number of declared data variables than either the ad hoc teams (AT) or the disciplined teams (DT).

(11) A slight trend existed for both the individuals (AI) and the disciplined teams (DT) to have a larger relative percentage of segment-global usage pairs (i.e., the ratio of actual segment-global usage pairs to possible segment-global usage pairs) than the ad hoc teams (AT) for nonentry global variables whose values were not modified during execution (i.e., the simplest kind of "named constants"), according to the (SEG,GLOBAL) USAGE RELATIVE PERCENTAGES\NONENTRY\ UNMODIFIED aspect.

(12) According to the STATEMENT TYPE COUNTS\IF and STATEMENT TYPE PERCENTAGE\IF aspects, both the individuals (AI) and the disciplined teams (DT) coded noticeably fewer IF statements than the ad hoc teams (AT), in terms of both total number and percentage of total statements.

(13) A trend existed, according to the STATEMENT TYPE COUNTS\ (PROC)CALL\INTRINSIC aspect, for the ad hoc teams (AT) to make

a larger number of calls on intrinsic procedures (i.e., built-in language-provided routines primarily for input-output) than either the individuals (AI) or the disciplined teams (DT).

(14) According to the STATEMENT TYPE COUNTS\RETURN aspect, the ad hoc teams (AT) had a noticeably larger number of RETURN statements than either the individuals (AI) or the disciplined teams (DT).

(15) According to the DECISIONS aspect, both the individuals (AI) and the disciplined teams (DT) tended to code fewer decisions (i.e., IF, WHILE, or CASE statements) than the ad hoc teams (AT).

(16) A trend existed for the ad hoc teams (AT) to have more calls to intrinsic procedures, with a noticeably larger number of calls to intrinsic routines (i.e., built-in language-provided procedures and functions, primarily for input-output and type conversion), than either the individuals (AI) or the disciplined teams (DT), according to the INVOCATIONS\PROCEDURE\INTRINSIC and INVOCATIONS\INTRINSIC aspects, respectively.

(17) According to the (SEG,GLOBAL,SEG) DATA BINDINGS\POSSIBLE aspect, there was a slight trend for both the individuals (AI) and the disciplined teams (DT) to have fewer possible data bindings [Stevens, Myers, and Constantine 74] (i.e., occurrences of the situation where a global variable r is both potentially modified by a segment p and potentially accessed by a segment q, with p different from q) than the ad hoc teams (AT).

(18) According to the COMPUTER JOB STEPS aspect, the disciplined teams (DT) required very noticeably fewer computer job steps (i.e., module compilations, program executions, or miscellaneous job steps) than both the individuals (AI) and the ad hoc teams (AT).

(19) This same difference was definitely apparent in the total number of module compilations, the number of unique (i.e., not an identical recompilation of a previously compiled module) module compilations, the number of program

executions, and the number of essential job steps (i.e., unique module compilations plus program executions), according to the COMPUTER JOB STEPS\MODULE COMPILATIONS, COMPUTER JOB STEPS\MODULE COMPILATIONS\UNIQUE, COMPUTER JOB STEPS\PROGRAM EXECUTIONS, and ESSENTIAL JOB STEPS aspects, respectively.

(20) A trend existed for both the individuals (AI) and the ad hoc teams (AT) to have required more miscellaneous job steps (i.e., auxiliary compilations or executions of something other than the final software product) than the disciplined teams (DT), according to the COMPUTER JOB STEPS\MISCELLANEOUS aspect.

(21) According to the AVERAGE UNIQUE COMPILATIONS PER MODULE and MAX UNIQUE COMPILATIONS F.A.O. MODULE aspects, respectively, the disciplined teams (DT) required fewer unique compilations per module on the average, with a definite trend toward fewer unique compilations for any one module in the worst case, than either the individuals (AI) or the ad hoc teams (AT).

(22) According to the LINES aspect, there was a definite trend for the individuals (AI) to have produced fewer total symbolic lines (includes comments, compiler directives, statements, declarations, etc.) than the disciplined teams (DT) who produced fewer than the ad hoc teams (AT).

(23) A definite trend existed for the individuals (AI) to have a larger relative percentage of segment-global usage pairs for entry globals and for entry globals which could be modified during execution than the disciplined teams (DT) who had a larger still percentage than the ad hoc teams (AT), according to (SEG,GLOBAL) USAGE RELATIVE PERCENTAGES\ENTRY and (SEG,GLOBAL) USAGE RELATIVE PERCENTAGES\ENTRY\MODIFIED aspects, respectively.

(24) According to the PROGRAM CHANGES aspect, there existed a trend for the disciplined teams (DT) to require fewer textual revisions to build and debug the software than the individuals (AI) who required fewer revisions than the ad hoc teams (AT).

The following numbered sentences simply provide English translations for the non-null dispersion conclusions presented in symbolic equation form in Table 2.2. They may be skimmed by the reader since they do not add to the information appearing in the table.

(1) The individuals (AI) displayed noticeably less variation in the number of formal parameters passed by reference than both the ad hoc teams (AT) and the disciplined teams (DT), with a similar trend in the percentage of reference parameters compared to the total number of declared data variable, according to the DATA VARIABLE SCOPE COUNTS\NONGLOBAL\ PARAMETER\REFERENCE and DATA VARIABLE SCOPE PERCENTAGES\ NONGLOBAL\PARAMETER\REFERENCE aspects.

(2) According to the PARAMETER PASSAGE TYPE PERCENTAGES\VALUE and PARAMETER PASSAGE TYPE PERCENTAGES\REFERENCE aspects, both the ad hoc teams (AT) and the disciplined teams (DT) tended to have more variation in the percentage of value parameters and reference parameters compared with the total number of formal parameters declared than the individuals (AI).

(3) The individuals (AI) had less variation in the number of possible segment-global usage pairs (i.e., potential access of a global variable by a routine) involving nonentry globals than either the ad hoc teams (AT) or the disciplined teams (DT), according to the (SEG,GLOBAL) POSSIBLE USAGE PAIRS\ NONENTRY aspect.

(4) According to the (SEG,GLOBAL,SEG) DATA BINDINGS\ACTUAL\ INDEPENDENT aspect, there was a very slight trend for the individuals (AI) to have less variation in the number of actual data bindings [Stevens, Myers, and Constantine 74] (i.e., occurrences of the situation where a global variable r is both actually modified by a segment p and actually accessed by a segment q, with p different from q) in which the two routines were "independent" (i.e., neither segment can invoke the other, directly or indirectly) than both the ad hoc teams (AT) and the disciplined teams (DT).

(5) The individuals (AI) exhibited noticeably greater variation

than either the ad hoc teams (AT) or the disciplined teams (DT) in the number of miscellaneous job steps (i.e., auxiliary compilations or executions of something other than the final software project), according to the COMPUTER JOB STEPS\MISCELLANEOUS aspect.

(6) In the number of calls in general and of calls to programmer-defined routines in particular, the individuals (AI) displayed noticeably greater variation than both the ad hoc teams (AT) and the disciplined teams (DT), according to the INVOCATIONS and INVOCATIONS\NONINTRINSIC aspects.

(7) According to the STATEMENTS aspect, a very slight trend existed for the ad hoc teams (AT) to show less variability than either the disciplined teams (DT) or the individuals (AI) in the number of executable statements.

(8) A trend existed for both the individuals (AI) and the disciplined teams (DT) to have greater variability than the ad hoc teams (AT) in the average (per function) number of calls to programmer-defined functions, according to the AVG INVOCATIONS PER (CALLED) SEGMENT\FUNCTION aspect.

(9) According to the (SEG,GLOBAL) ACTUAL USAGE PAIRS\MODIFIED aspect, a definite trend existed for the ad hoc teams (AT) to have less variability than either the individuals (AI) or the disciplined teams (DT) in the number of actual segment-global usage pairs (i.e., actual access of a global variable by a routine) involving globals which were modified during execution.

(10) According to the AVERAGE SEGMENTS PER MODULE aspect, the individuals (AI) and the disciplined teams (DT) both exhibited noticeably less variation in the average number of routines per module than the ad hoc teams (AT).

(11) The ad hoc teams (AT) were noticeably more variable than either the disciplined teams (DT) or the individuals (AI) in the percentage of coded RETURN statements compared with the total number of statements, according to the STATEMENT TYPE PERCENTAGES\RETURN aspect.

(12) According to the AVERAGE GLOBAL VARIABLE PER MODULE\MODIFIED aspect, the ad hoc teams (AT) displayed a definite trend

toward greater variability than both the individuals (AI) and the disciplined teams (DT) in the average number of globals per module which were modified during execution.

(13) The individuals (AI) and the disciplined teams (DT) were both noticeably less variable than the ad hoc teams (AT) in the number of possible segment-global usage pairs where the global variable was nonentry and modified during execution, according to the (SEG, GLOBAL) POSSIBLE USAGE PAIRS\NONENTRY\MODIFIED aspect.

(14) According to the (SEG,GLOBAL,SEG) DATA BINDINGS\POSSIBLE aspect, the ad hoc teams (AT) tended toward greater variability than either the individuals (AI) or the disciplined teams (DT) in the number of possible data bindings.

(15) According to the STATEMENT TYPE COUNTS\(PROC)CALL, STATEMENT TYPE COUNTS\(PROC)CALL\NONINTRINSIC, INVOCATIONS\PROCEDURE, and INVOCATIONS\PROCEDURE\NONINTRINSIC aspects, both the individuals (AI) and the ad hoc teams (AT) were noticeably more variable than the disciplined teams (DT) in the number of calls to intrinsic and nonintrinsic procedures, with a similar trend for calls to nonintrinsic procedures alone.

(16) This same difference appeared in the average number of intrinsic procedure calls per calling segment, according to the AVG INVOCATIONS PER (CALLING) SEGMENT\PROCEDURE\INTRINSIC aspect.

(17) According to the DATA VARIABLES SCOPE PERCENTAGES\GLOBAL\NONENTRY\MODIFIED aspect, the disciplined teams (DT) displayed noticeably smaller variation than either the individuals (AI) or the ad hoc teams (AT) in the percentage of nonentry global variables that were modified during execution compared to the total number of data variables declared.

(18) According to the AVERAGE TOKENS PER STATEMENT aspect, a definite trend existed for the disciplined teams (DT) to exhibit greater variability in the average number of tokens (i.e., basic symbolic units) per statement than both the individuals (AI) and the ad hoc teams (AT).

(19) The trend toward less variation among both the individuals

(AI) and the ad hoc teams (AT) than among the disciplined teams (DT) existed in the number of global variables and in the number of formal parameters, according to the DATA VARIABLE SCOPE COUNTS\GLOBAL and DATA VARIABLE SCOPE COUNTS\NONGLOBAL\PARAMETER aspects, respectively.

(20) A similar difference in variability existed noticeably in the percentages, compared to the total number of declared data variables, of globals, of nonglobals, of formal parameters, and of formal parameters passed by value, according to the DATA VARIABLE SCOPE PERCENTAGES\GLOBAL, DATA VARIABLE SCOPE PERCENTAGES\NONGLOBAL, DATA VARIABLE SCOPE PERCENTAGES\NONGLOBAL\PARAMETER, and DATA VARIABLE SCOPE PERCENTAGES\NONGLOBAL\PARAMETER\VALUE aspects, respectively.

(21) According to the (SEG,GLOBAL) POSSIBLE USAGE PAIRS and (SEG,GLOBAL) POSSIBLE USAGE PAIRS\NONENTRY\UNMODIFIED aspects, there was a noticeable difference in variability, with the individuals (AI) less than the disciplined teams (DT) less than the ad hoc teams (AT), for the total number of possible segment-global usage pairs, with a similar trend for possible usage pairs in which the global variable was nonentry and not modified during execution.

(22) There was a noticeable difference in variability, with the disciplined teams (DT) less than the individuals (AI) less than the ad hoc teams (AT), in the maximum number of unique compilations for any one module, according to the MAX UNIQUE COMPILATIONS F.A.O. MODULE aspect.

(23) According to the STATEMENT TYPE COUNTS\RETURN aspect, there was a difference in variability, with the disciplined teams (DT) less than the individuals (AI) less than the ad hoc teams (AT), for the number of RETURN statements coded.

Appendix 3. English Paraphrase of Relaxed Differentiation Analysis

The following two paragraphs simply provide an English paraphrase of the "relaxed differentiation" details presented in Tables 4.1 and 4.2, respectively.

On location comparisons, four programming aspects yielded completely differentiated conclusions. They are "relaxed" to partially differentiated conclusions as follows:

1. From DT < AI < AT on PROGRAM CHANGES, the DT < AI = AT conclusion overwhelmingly dwarfs the DT = AI < AT conclusion
2. The DT < AT difference is more pronounced than the AI < DT difference from AI < DT < AT on LINES
3. AT < DT < AI on (SEG,GLOBAL) USAGE RELATIVE PERCENTAGES\ENTRY is more significantly "relaxed" to AT < DT = AI than to AT = DT < AI
4. The AT < DT and DT < AI differences from AT < DT < AI on (SEG,GLOBAL) USAGE RELATIVE PERCENTAGES\ENTRY\MODIFIED are both exactly equally strong

On dispersion comparisons, three programming aspects yielded completely differentiated conclusions. They are "relaxed" to partially differentiated conclusions as follows:

1. The DT < AI difference is much more pronounced than the AI < AT difference from DT < AI < AT on MAX UNIQUE COMPILATIONS F.A.O. MODULE
2. From DT < AI < AT on STATEMENT TYPE COUNTS\RETURN, the DT = AI < AT conclusion overwhelmingly dwarfs the DT < AI = AT conclusion
3. AI < DT < AT on (SEG,GLOBAL) POSSIBLE USAGE PAIRS is more significantly "relaxed" to AI < AT = DT than to DT = AI < AT
4. The AI < DT difference is more pronounced than the DT < AT difference from AI < DT < AT on (SEG,GLOBAL) POSSIBLE USAGE PAIRS\NONENTRY\UNMODIFIED

Appendix 4. English Categorization of Directionless Distinctions

The following two paragraphs provide a complete itemization of directionless distinctions. The information contained in Tables 2 and 4 has simply been reorganized and presented in English to support a directionless view of the study's results.

Specifically, for the study's location comparisons:

(1) The distinction

AI (individuals) ≠ AT (ad hoc teams) = DT (disciplined teams)

was observed for none of the process aspects and for several product aspects, including

- the raw count of programmer-defined segments (i.e., routines),
- the raw count of programmer-defined data variables,
- several raw counts and relative percentages of data variables according to their scope (i.e., global, parameter, or local),
- the raw count of potential segment-global usage pairs (which is strongly dependent on the raw counts of segments and globals, both of which are also in this category), and
- several "per segment" averages of other raw counts (i.e., formal parameters, executable statements, and nonintrinsic calls).

(2) The distinction

AT (ad hoc teams) ≠ DT (disciplined teams) = AI (individuals)

was observed for none of the process aspects and for several product aspects, including

- the raw count of lines of symbolic source code,
- both the raw count and relative percentage of IF statements,
- the raw count of programmed decisions (i.e., total number of IF, CASE, and WHILE statements),
- the raw count of RETURN statements,
- the raw counts of calls to intrinsic routines and intrinsic

procedures,
- one ratio of actual to possible accessibility of globals by segments, and
- the raw count of possible communication paths between segments via globals.

(3) The distinction

DT (disciplined teams) ≠ AI (individuals) = AT (ad hoc teams)

was observed for nearly all the process aspects, including
- nearly all the raw counts of computer job steps, including both the total count and all the subclassification counts (i.e., compilations, executions, miscellaneous), except for identical compilations,
- both "per module" counts of unique compiles, the average and the (worst case) maximum, and
- the amount of revision and change made to the source code during development,

but for none of the product aspects.

Specifically, for the study's dispersion comparisons:

(1) The distinction

AI (individuals) ≠ AT (ad hoc teams) = DT (disciplined teams)

was observed for one process aspect, namely
- the raw count of miscellaneous computer job steps (i.e., auxiliary compilations or executions of something other than the final product),

and for several product aspects, including
- the raw count and several relative percentages of reference parameters,
- a few raw counts of potential segment-global usage pairs,
- the raw count of total invocations and invocations of programmer-defined routines, and
- the raw count of actual segment-global-segment data bindings in which neither segment could invoke the other.

(2) The distinction

AT (ad hoc teams) ≠ DT (disciplined teams) = AI (individuals)

was observed for none of the process aspects and for several

product aspects, including

- two "per module" averages of other raw counts, (i.e., segments and global variables which were modified during execution),
- the raw count of executable statements,
- both the raw count and relative percentage of RETURN statements,
- the average number of calls made to programmer-defined segments which were functions rather than procedures,
- the raw count of actual segment-global usage pairs in which the global variable is modified during execution,
- the raw count of potential segment-global usage pairs in which the global variable is not accessible across modules and is modified, and
- the raw count of potential segment-global-segment data bindings.

(3) The distinction

DT (disciplined teams) ≠ AI (individuals) = AT (ad hoc teams)

was observed for one process aspect, namely

- the (worst case) maximum count of unique compiles for any one module,

and for several product aspects, including

- several raw counts and relative percentages of data variables according to their scope (i.e., global, parameter, or local),
- the raw counts of calls to procedures and to programmer-defined procedures,
- the average number of calls to built-in procedures per calling segment, and
- the average number of tokens per statement.