A Controlled Experiment Quantitatively Comparing Software Development Approaches

VICTOR R. BASILI AND ROBERT W. REITER, JR., MEMBER, IEEE

Abstract—A software engineering research study has been undertaken to empirically analyze and compare various software development approaches; its fundamental features and initial findings are presented in this paper. An experiment was designed and conducted to confirm certain suppositions concerning the beneficial effects of a particular disciplined methodology for software development. The disciplined methodology consisted of programming teams employing certain techniques and organizations commonly defined under the umbrella term structured programming. Other programming teams and individual programmers both served as control groups for comparison. The experimentally tested hypotheses involved a number of quantitative, objective, unobtrusive, and automatable measures of programming aspects dealing with the software development process and the developed software product. The experiment's results revealed several programming aspects for which statistically significant differences existed between the disciplined methodology and the control groups. The results were interpreted as confirmation of the original suppositions and evidence in favor of the disciplined methodology. This paper describes the specific features of the experiment; outlines the investigative approach used to plan, execute, and analyze it; reports its immediate results; and interprets them according to intuitions regarding the disciplined methodology.

Manuscript received June 30, 1979; revised January 15, 1980. This work was supported in part by the Air Force Office of Scientific Research under Grant AFSOR-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. At the time this work was done, both authors were with the University of Maryland.

V. R. Basili is with the Department of Computer Science, University of Maryland, College Park, MD 20742.

R. W. Reiter, Jr. was with the Department of Computer Science, University of Maryland, College Park, MD 20742. He is now with the Software Engineering and Technology Department, IBM Federal Systems Division, Bethesda, MD 20034.

Index Terms—Controlled experimentation, empirical study, programming measurement, programming methodology, programming teams, software development, software metrics, structured programming practices.

I. INTRODUCTION

MUCH has been written about methodologies for developing computer software (e.g., [9], [11], [15], [17], [20], [28]). Most of these methodologies are founded on sound logical principles. Case studies have occasionally been conducted to demonstrate their effectiveness (e.g., [1], [6]), Their adoption within production ("real-world") environments has generally been successful. Having practiced adaptations of these methodologies, software designers and programmers have often asserted qualitatively that they got the job done faster, made fewer errors, or produced a better product (e.g., [12]). Unfortunately, solid empirical evidence that comparatively and quantitatively assesses any particular methodology is scarce (e.g., [18], [21], [23], [24]). This is due partially to the cost and impracticality of a valid experimental setup within a production environment.

Thus the question remains, are measurable benefits derived from programming methodologies, with respect to either the software development process or the developed software product? Even if the perceived benefits are real, it is not clear that they can be quantified or monitored, in order to confirm the effectiveness of the methodologies. Software development is still too artistic, in the aesthetic or sponta-

0098-5589/81/0500-0299$00.75

neous sense. In order to understand it more fully, manage it more cost-effectively, and adapt it more readily to challenging applications or situations, software development must become more scientific, in the engineered and deliberate sense. More empirical study, data collection, and experimental analysis are required to achieve this goal.

The purpose of the research reported in this paper is 1) to quantitatively investigate the effect of methodologies and programming environments on software development and 2) to develop an investigative methodology based on scientific experimentation and tailored to this particular application. It involves the measurement and analysis of both the software process and the software product in a manner which is minimally obtrusive (to those developing the software), objective, and automatable. The goal of the research was to verify the effectiveness of a particular programming methodology and to identify various quantifiable aspects that could demonstrate such effectiveness.

To this end, a controlled experiment was conducted involving several replications of a specific software development task under varying programming environments. The experiment compared three distinct groupings of software developers: individual programmers, three-person programming teams, and three-person programming teams using a disciplined methodology. The disciplined methodology consisted of an integrated set of software development techniques and team organizations, including top-down design, process design language, structured programming, code reading, and chief programmer teams.

The study examines differences in the expectancy of software development behavior under the programming environments represented by these groups. The basic premise is that distinctions among the groups exist both in the *process* and in the *product*. With respect to the software development process, a disciplined team should have advantages over both an individual and an ordinary team, displaying superior performance on cost factors such as computer usage and number of errors made. This is because of the discipline itself and because of the ability to use team members as resources for validation. With respect to the developed software product, it is believed that a disciplined team should approximate an individual with regard to design and source code characteristics (such as decision structure and global data accessibility) and at the very least lie somewhere between an individual and an ordinary team. This is because the disciplined methodology should enable the team to act as a mentally cohesive unit during the design, coding, and testing phases.

The study's findings reveal several programming characteristics for which statistically significant differences do exist among the groups and tend to support these basic premises.

The investigation has been conducted in a laboratory or proving-ground fashion, in order to achieve some statistical significance and scientific respectability without sacrificing production realism and professional applicability. By scaling down a typical production environment while retaining its important characteristics, the laboratory setting provides for a reasonable compromise between the extremes of

1) "toy" experiments, which can afford elaborate experimental designs and large sample sizes but often suffer from a basic task that is rather unrelated to production situations or involve a context from which it is difficult to extrapolate or scale up (e.g., introductory computer course students taking multiple-choice quizzes based on 30-line programs), and

2) "production" experiments, which offer a high degree of realism by definition but incur prohibitively high costs even for the simplest and weakest experimental designs (i.e., replication of a nontrivial programming project is clearly expensive).

The experiment in this study was conducted within an academic environment where it was possible to achieve an adequate experimental design and still simulate key elements of a production environment.

An initial phase of investigative effort has been completed and its prominent features are presented in the remainder of this paper. Section II gives details pertaining to the experiment itself. Section III describes the investigative methodology used to plan, execute, and analyze the experiment. Sections IV and V present the experiment's findings, segregated into empirical results (resulting from statistical analysis of the measurements) and intuitive judgements (resulting from interpretation of the empirical results), respectively. (Different statistical analyses and additional interpretations of the same experimental data have appeared in [5], [22] as explained below.) Section VI makes some concluding remarks and mentions further work planned for the study. Appendices I and II explain concisely what programming aspects were measured and contain the observed raw data scores.

It should be noted that the terms "methodology" and "methodological" (in reference to software development) 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.

II. Specifics

Experimental Design

The basic task involved in the experiment was the completion of a specific software development project. There were 19 replications of the basic task, each performed concurrently and independently by a separate software development "team." There were two experimental treatment factors (independent variables): size of the development "team" and degree of methodological discipline. For each factor, there were two experimental treatment factor levels: for the size factor, a single individual and a three-person team; for the degree-of-discipline factor, an ad hoc approach and a disciplined methodology.

The experiment was embedded within two academic courses, and every student enrolled in those courses participated in the experiment. Development "teams" were formed among the subjects: in one course, the students were allowed to choose between segregating themselves as individual programmers or combining with two other classmates as three-person programming teams; in the other course, the students were assigned (by the researchers) into three-person teams. The experiment was designed in this manner because the two academic courses themselves provided the two levels of the second experimental treatment factor. This scheme yielded three groups of 6, 6, and 7 "teams," designated AI, AT, and

DT, respectively. Each group was exposed to a particular combined factor-level treatment according to the following partial factorial arrangement:

(AI) single individuals using an ad hoc approach,
(AT) three-person teams using an ad hoc approach, and
(DT) three-person teams using a particular disciplined methodology.

A set of experimental observations (dependent variables), composed of 35 programming aspects related to the development process and the software product, had been identified prior to conducting the experiment. The performance of each development "team" was quantified according to each programming aspect. The overall experiment thus technically consisted of a series of simultaneous univariate experiments, one for each observed programming aspect, all sharing a common experimental design and a common raw data sample.

Although this experimental design basically followed the reductionist paradigm, in which most variables are controlled so that the relationships among the remaining few can be isolated, the ideal was only approximated. Specifically, there were two variables which the design did not explicitly control: the personal ability/experience of the participants and the amount of actual time/effort they devoted to the project. These variables could only be allowed to vary among the groups in what was assumed to be a random manner. However, information from a pretest questionnaire was used to balance the personal ability/experience of the group DT participants (only) across those seven teams. As a reasonable measure of individual programmer skill levels, the participants' grades from a pertinent prerequisite course provided a post-experimental confirmation that programming ability was fairly evenly distributed among the groups.

Software Development Methodologies

The disciplined methodology imposed on teams in 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, walkthroughs, and chief programmer team organization. These were taught as an integral part of the course that the subjects were taking, and the course material was organized around [2], [9], [17] as textbooks. Since the subjects were novices in the methodology, they executed the techniques and organizations to varying degrees of thoroughness and were not always as successful as seasoned users of the methodology would be.

Specifically, the disciplined methodology prescribed the use of a PDL for expressing the design of the problem solution. The design was expressed in a top-down manner, each level representing a solution to the problem at a particular level of abstraction and specifying the functions to be expanded at the next level. The PDL consisted of a specific set of structured control and data structures, plus an open-ended designer-defined set of operators and operands corresponding to the level of the solution and the particular application. Design and code reading involved the critical review of each team member's PDL or code by at least one other member of the team. Walk-throughs represented a more formalized presentation of an individual's work to the other team members in which the PDL or code was explained step by step. Under the chief programmer team organization, one team member was responsible for designing and refining the top-level solution to the problem in PDL, identifying system components to be implemented, defining their interfaces, and implementing the key code; the other team members were each responsible for designing or coding various system components, as assigned by the chief programmer. Responsibility for librarian activities (entering or revising code stored on-line, making test runs, etc.) was allocated among the three team members in the manner most comfortable for them.

Each individual or team in groups AI and AT was allowed to develop the software in a manner entirely of their own choosing, which is herein referred to as an ad hoc approach. No methodology was taught in the course these subjects were taking. Informal observation by the researchers confirmed that approaches used by the individuals and ad hoc teams were indeed lacking in discipline and did not utilize the key elements of the disciplined methodology (e.g., an individual working alone cannot practice code reading, and it was evident that the ad hoc teams did not use a PDL or formally do a top-down design).

Programming Environment

Several particulars of the experimental programming environment contribute significantly to the context in which the experiment's results must be appraised. These include the setting in which the experiment was conducted, the software development project that served as the experimental task, the people who participated as subjects, the computer system access mode they used, and the computer programming language in which the software was written.

The experiment was conducted during the Spring 1976 semester, January through May, within regular academic courses given by the Department of Computer Science on the College Park campus of the University of Maryland. Two comparable advanced elective courses were utilized, each with the same academic prerequisites. The experimental task and treatments were built into the course material and assignments. 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 compiler for a small high-level language and a simple stack machine; it involved string processing and language translation (via scanning, parsing, code generation, and symbol table management). The total task was to design, implement, test, and debug the complete computer software 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." Each "team" worked independently to build its own system, using the same specifications, computer resource

allocation, calendar time allotment, implementation language, debugging tools, etc. The delivered systems each passed an independent acceptance test.

The participants were advanced undergraduate and graduate students in the Department of Computer Science, a few with as much as three years' 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. A reasonable degree of homogeneity seemed to exist among the participants with respect to personal factors such as ability/experience, motivation, time/effort devoted to the project, etc. If anything, based on the researchers' subjective judgment, the participants in groups AI and AT seemed to have a slight edge over those in group DT with respect to native programming ability and formal training in the application area.

The host computer system used by all "teams" was a Univac 1100 machine with the usual Exec operating system, supporting both batch and interactive access. It was observed that almost all "teams" consistently preferred the interactive access mode; only one of AI "teams" used the batch access mode extensively.

The implementation language was the high-level, structured-programming language SIMPL-T [7], taught and used extensively in regular course work at the University. SIMPL-T contains the following control constructs: sequence, ifthen, ifthenelse, whiledo, case, exit from loop, and return from routine (but no goto). SIMPL-T allows basically 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

During the course of the experiment, 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 an historical database, consisting of all source code and test data accumulated throughout the project development period, for each development "team." The raw information in this database was subsequently reduced to obtain the experimental observations. The final products were isolated from the database and measured for various syntactic and organizational aspects of the finished product source code. Effort and cost measurements were also extracted from the database. The inputs to the analysis, in the form of scores for the various programming aspects, reflect the quantitatively measured character of the product and effort of the process. (These raw data scores are presented in Appendix II.) Much of this data reduction was done automatically within a specially instrumented compiler. The same collection and reduction mechanism was uniformally applied to all development "teams," ensuring the objectivity of the observations and measurements.

Programming Aspects and Metrics

The dependent variables studied in this experiment are called programming aspects. They represent specific isolatable and observable features of programming phenomena. Furthermore, they are measured in a manner that may be characterized as quantitative (on at least an interval scale [10, pp. 65-67], objective (without inaccuracy due to human subjectivity), unobtrusive (to those developing the software), and automatable (not depending on human agency).

The variables fall into two categories: process aspects and product aspects. 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 itself. Examples of product aspects are number of lines, frequency of particular statement types, average size of data variables' scope, etc. For each programming aspect there exists an associated metric, a specific algorithm which ultimately defines that aspect and by which it is measured.

Table I lists the particular programming aspects examined in this investigation. They appear grouped by category, with indented qualifying phrases to specify particular variants of certain general aspects. When referring to an individual aspect, a concatenation of the heading line with the qualifying phrases (separated by \ symbols) is used; for example, COMPUTER JOB STEPS\MODULE COMPILATION\UNIQUE denotes the number of COMPUTER JOB STEPS that were MODULE COMPILATIONS in which the source code was UNIQUE from all other compiled versions. Explanatory notes (keyed to the list in Table I) about the programming aspects are given in Appendix I, with definitions for the nontrivial or unfamiliar metrics. Technical meanings for various system- or language-dependent terms (e.g., module, segment) also appear there. Since computer programming terminology is not particularly standardized, the reader is cautioned against drawing inferences not based on this paper's definitions.

The programming aspects had been consciously planned in advance of collecting and extracting data because intuition suggested that they would serve well as quantitative indicators of important qualitative characteristics of software development phenomena. It was predicted a priori that these so-called "confirmatory" aspects would verify the study's basic premises regarding the programming methodologies being investigated.

The overall study also examined many so-called "exploratory" programming aspects: measurements which could be collected and extracted cheaply (even as a natural by-product sometimes) along with the "confirmatory" aspects, but for which there was little serious expectation that they would be useful indicators of differences among the groups. They were included in the overall study with the intent of observing as many aspects as possible on the off chance of discovering

any unexpected tendency or difference, thus combining elements of both confirmatory and exploratory data analysis within one common experimental setting [27]. For these "exploratory" programming aspects and their results, interested readers are referred to [5], [22].

III. Approach

The investigative methodology can be characterized as an empirical study based on the "construction" paradigm in which multiple subjects are closely monitored during actual "production" experiences, each subject performing the same task, with controlled variation in specific variables. It uses scientific experimentation and statistical analysis based on a "differentiation among groups by aspects" paradigm in which possible differences among the groups, as indicated by differences in certain quantitatively measured aspects of the observed phenomena, are the target of the analysis. This use of "difference discrimination" as the analytical technique dictates a model of homogeneity hypothesis testing that influences nearly every element (or step) of the methodology.

Fig. 1, the approach schematic, charts some of the relationships among the various steps of the investigative methodology. The remainder of this section outlines the approach by briefly defining each step and discussing how it was applied in the research effort at hand.

Step 1–Questions of Interest: Several questions of interest were initiated and refined so that answers could be given in the form of statistical conclusions and research interpretations. The final questions of interest culminated in the form "during software development, what comparisons between the effects of the three factor-level combinations a) single individuals, b) ad hoc teams, and c) disciplined teams appear as differences in the various quantitatively measurable aspects of the software development process and product? Furthermore, what kind of differences are exhibited and what is the direction of these differences?"

Step 2–Research Hypotheses: Based upon the questions of interest, precise research hypotheses were formulated as disjoint pairs designated null and alternative, to be supported or refuted by the evidence.

A precise meaning was given to the notion "what kind of difference." In order to address the expectency of behavior under the experimental treatments, the investigation focused on differences in central tendency or average value of the quantifiable programming aspects. These "location" comparisons and their results are the topic of this paper. The overall study also addressed the predictability of behavior under the experimental treatments by considering differences in variability around the central tendency of observed values of the programming aspects. For these "dispersion" comparisons and their results, interested readers are referred to [5], [22].

The schema for the research hypotheses may be stated as follows. "In the context of a one-person do-able software development project, there $<$ is not $|$ is $>$ a difference in the location of the measurements on programming aspect $<X>$ between individuals (AI), ad hoc teams (AT), and disciplined teams (DT)." For each programming aspect "X" in the set under consideration, this schema generates a pair of nondirectional research hypotheses, depending upon the selection of "is not" or "is" corresponding to the null and alternative hypothesis.

Step 3–Statistical Model: The choice of a statistical model makes explicit various assumptions regarding the experimental design, the dependent variables, the underlying population distributions, etc. Because the study involves a homogeneity-of-populations problem with shift alternative, the multisample model used here requires the following criteria: independent populations, independent and random sampling within each population, continuous underlying distributions for each population, homoscedasticity (equal variances) of underlying distributions, and interval scale of measurement [10, pp. 65-67] for each programming aspect. Although random sampling was not explicitly achieved in this study by rigorous sampling procedures, it was nonetheless assumed on the basis of the apparent representativeness of the subject pool and the lack of obvious reasons to doubt otherwise. Due to the small sample sizes and the unknown shape of the underlying distributions, a nonparametric statistical model was used.

Whenever statistics is employed to "prove" that some systematic effect–in this case, a difference among the groups–exists, it is important to measure the risk of error. This is usually done by reporting a significance level α [10, p. 79], which represents the probability of deciding that a systematic effect exists when in fact it does not. In the model, the hypothesis testing for each programming aspect was regarded as a separate independent experiment. Consequently, the significance level is controlled and reported experimentwise (i.e., per aspect). While the assumption of independence between such experiments is not entirely supportable, this procedure is valid as long as statistical inferences that couple two or more of the programming aspects are avoided or properly qualified.

Step 4–Statistical Hypotheses: The research hypotheses must be translated into statistically tractable form, called statistical hypotheses. In this study, the research hypotheses are concerned with directional differences among three programming environments. Since the corresponding mathematical statements are not directly tractable, they were broken down into the set of four statistical hypotheses pairs shown below. The hypotheses pair

$$\text{null: } AI = AT = DT \quad \text{alternative: } \sim(AI = AT = DT)$$

addresses the existence of an overall difference among the groups. The hypotheses pairs

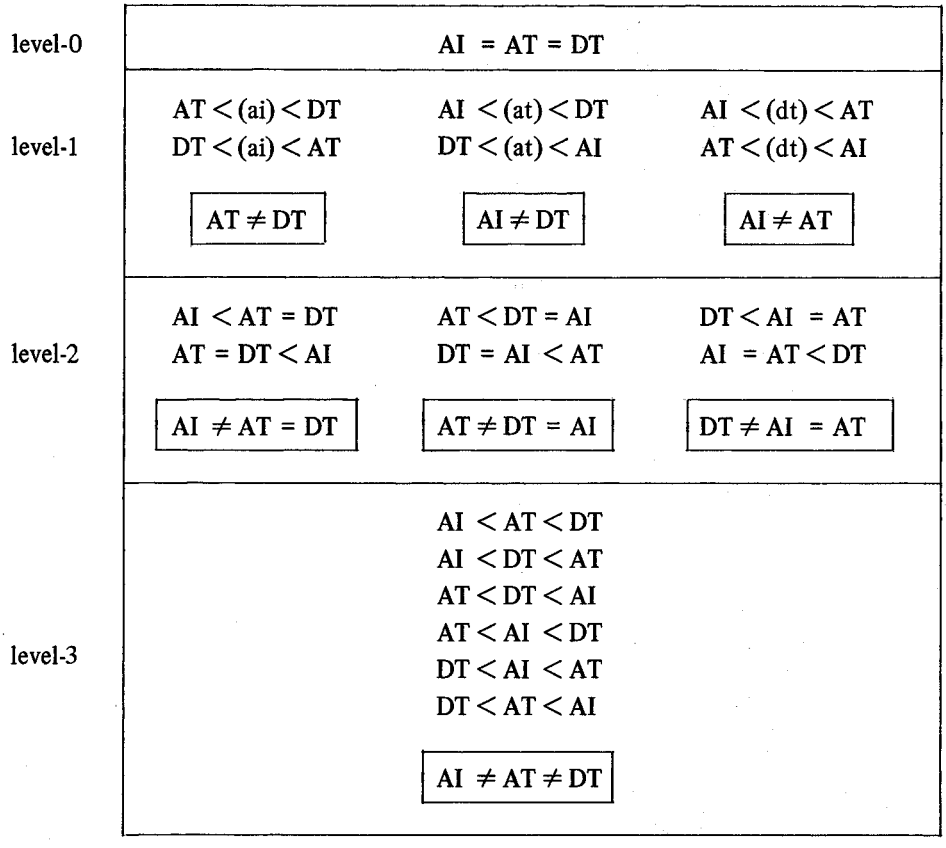
$$\begin{aligned}
&\text{null: } AI = AT && \text{alternative: } AI \neq AT \text{ or } AI < AT \text{ or } AT < AI \\
&\text{null: } AT = DT && \text{alternative: } AT \neq DT \text{ or } AT < DT \text{ or } DT < AT \\
&\text{null: } AI = DT && \text{alternative: } AI \neq DT \text{ or } AI < DT \text{ or } DT < AI
\end{aligned}$$

address the existence and direction of pairwise differences between groups. The results of these pairwise comparisons were used to explicate the overall comparison.

Thus, for any particular programming aspect, the research hypotheses pair corresponds to four different pairs (null and alternative) of scientific hypotheses. The results of testing each set of four hypotheses must be abstracted and organized into one statistical conclusion using the first research framework discussed in the next step.

Step 5–Research Frameworks: The research frameworks provide the necessary organizational basis for abstracting and conceptualizing the volume of statistical hypotheses (and statistical results that follow) into a smaller and more intellectually manageable set of conclusions. Two separate research frameworks have been chosen: 1) the framework of possible overall comparison outcomes for a given programming aspect and 2) the framework of general beliefs regarding expected effects of the experimental treatments on the comparison outcomes for the entire set of programming aspects. The first framework is employed in the statistical conclusions step because it can be applied in a statistically tractable manner, while the second framework is reserved for the research interpretations step since it is not statistically tractable and involves subjective judgment.

Since a finite set of three different programming environments (the AI, AT, and DT groups) are being compared, there exists a finite set of nineteen possible overall comparison outcomes for each aspect considered, as displayed in the following chart:



The level number associated (in the chart) with each outcome "equation" is exactly the number of statistically significant (pairwise) differences implied by or stated in that equation.

The level-0 equation indicates no distinction among the three groups. The level-1 equations indicate a difference between the two extreme groups, with the third group (designated in lowercase letters within parentheses) lying in between. The level-2 equations indicate that one group is different from each of the other two, while the level-3 equations indicate that all three groups are differentiated from one another. The equations appearing in boxes provide a direction-free "summary" of the corresponding set of equations. These 19 possible overall comparison outcomes comprise the first research framework and may be viewed as providing a complete "answer space" for the questions of interest. This framework is the basis for organizing and condensing the four statistical results into one statistical conclusion for each programming aspect considered.

The design of the experiments, the choice of treatment factors, etc., were partially motivated by certain general beliefs regarding software development, such as "disciplined methodology reduces software development costs." The implications, relative to these beliefs, of the possible outcomes of each aspect's experiment provide a second research framework. This framework is the basis for interpreting the study's findings in terms of evidence in favor of the general beliefs; details are given in Section V, Interpretation.

The overall study also employed a third research framework, based on abstracting what the study's findings indicate about certain higher level programming issues (such as data variable organization or intersegment communication). For this third framework and the corresponding interpretation, interested readers are referred to [5], [22].

Step 6–Experimental Design: The experimental design is

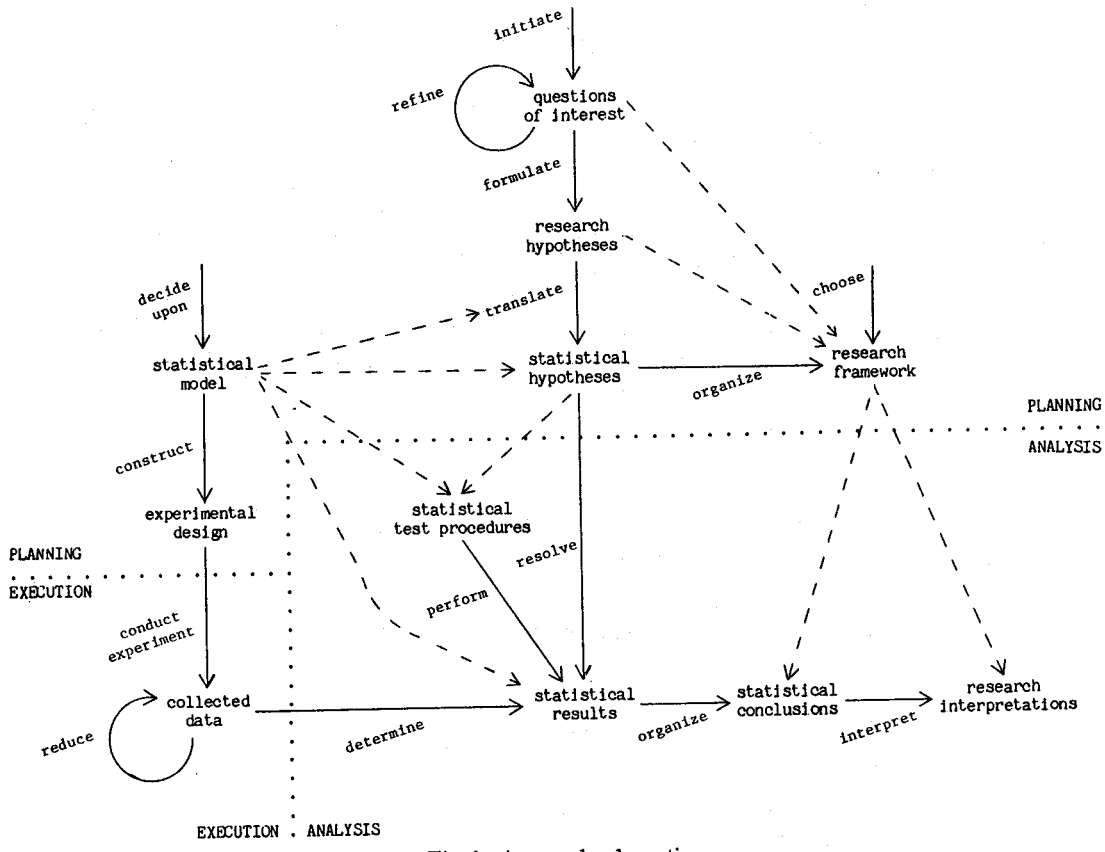


Fig. 1. Approach schematic.

the plan or setup according to which the experiment is actually conducted or executed. It is based upon the statistical model, and deals with practical issues such as experimental units, treatment factors and levels, experimental local control, etc. The experimental design employed for this study has been discussed in Section II, Specifics.

Step 7–Collected Data: The pertinent data to carry out the experimental design are collected and processed to yield the information to which the statistical test procedures were applied. Some details of this execution phase have been given in Section II, Specifics. The data themselves are listed in Appendix II.

Step 8–Statistical Test Procedures: As dictated by the statistical model, the statistical tests used in the study were nonparametric tests of homogeneity of populations against shift alternatives for small samples. In particular, the standard Kruskal-Wallis H-test [25, pp. 184-193] and Mann-Whitney U-test [25, pp. 116-127] were employed in the statistical results step. Ryan's method of adjusted significance levels [16, pp. 97, 495-497], a standard procedure for controlling the experimentwise significance level when several tests are performed on the same scores as one experiment, was also employed in the statistical conclusions step. As part of Ryan's method, the rank means within the groups were used *a posteriori* to determine the direction of significant differences.

The critical level $\hat{\alpha}$ [10, p. 81] is defined as the minimum significance level at which the statistical test procedure would allow the null hypothesis to be rejected (in favor of the alternative) for the given sample data. It is a concise standardized way to state the full result of any statistical test procedure. A decision to reject the null hypothesis and accept the alternative is mandated if the critical level is low enough to be tolerated; otherwise a decision to retain the null hypothesis is made.

A different statistical analysis has been performed [5], [22], which postulated directional alternative hypotheses (and used one-tailed tests). Taking a slightly more conservative tack, this present paper makes no *a priori* assumptions regarding direction of observed differences (and uses two-tailed tests). It should be noted that, since the study's *a priori* general beliefs (see Section V, Interpretation) did involve differences in particular directions, some justification exists for using one-tailed tests in the statistical analysis. This would roughly halve the critical levels shown throughout this paper. However, results based on two-tailed tests are presented herein in order to avoid any objections concerning statistical technique.

Step 9–Statistical Results: For each pair of statistical hypotheses, there is one statistical result consisting of four components: 1) the null hypothesis itself; 2) the alternative hypothesis itself; 3) the critical level, stated as a probability value between 0 and 1; and 4) a decision either to retain the null hypothesis or to reject it in favor of (i.e., accept) the alternative hypothesis.

By convention, the null hypothesis purports that no systematic difference appears to exist, and the alternative hypothesis purports that some systematic difference seems to exist. The critical level is associated with erroneously accepting the alternative hypothesis (i.e., claiming a systematic difference when none in fact exists). The decision to retain or reject is reached on the basis of some tolerable level of significance, with which the critical level is compared to see if it is low

enough. In cases where a null hypothesis is rejected, the appropriate directional alternative hypothesis (if any) is given to indicate the direction of the systematic difference.

Conventional practice is to fix an arbitrary significance level (e.g., 0.05 or 0.01) in advance, to be used as the tolerable level; critical levels then serve only as stepping-stones toward reaching decisions and are not reported. For this study, it was deemed more appropriate to fix a tolerable level only for the purpose of a screening decision (simply to purge those results with intolerably high critical levels) and to explicitly retain a surviving critical level with each statistical result. The tolerable level of significance used throughout this study to screen critical levels was fixed at under 0.20. A critical level of 0.20 means that the odds of obtaining test scores exhibiting the same degree of difference, due to random chance fluctuations alone, are one in five.

As an example, the four statistical results for the programming aspect STATEMENT TYPE COUNTS\IF are shown below.

null hypothesis	alternative hypothesis	critical level	(screening) decision
AI = AT = DT	~ (AI = AT = DT)	0.063	reject
AI = AT	AI < AT	0.139	reject
AI = AT	AI ≠ DT	>0.999	retain
AT = DT	DT < AT	0.066	reject

Observe that the stated decisions reflect the application of the 0.20 tolerable level to the stated critical levels. Results under more stringent levels of significance can easily be determined by simply applying a lower tolerable level to form the decisions, e.g., at the 0.10 significance level, only the AI = AT = DT and AT = DT null hypothesis would be rejected.

Step 10–Statistical Conclusions: The volume of statistical results are organized and condensed into statistical conclusions according to the prearranged research framework(s). Specifically, the first research framework mentioned above was employed to reduce the four statistical results (with four individual critical levels) for each programming aspect to a single conclusion (with one overall critical level) for that aspect. The statement portion of a statistical conclusion is simply one of the nineteen possible overall comparison outcomes. Each overall comparison outcome is associated with a particular set of statistical results whose outcomes support the overall comparison outcome in a natural way. For example, the DT = AI < AT conclusion is associated with the following results:

reject AI = AT = DT in favor of ~ (AI = AT = DT),
reject AI = AT in favor of AI < AT,
retain AI = DT, and
reject AT = DT in favor of DT < AT.

Continuing the example started in Step 9, the statistical results shown there for the STATEMENT TYPE COUNTS\IF aspect are reduced to the statistical conclusion DT = AI < AT with 0.139 critical level overall. The four results match those associated above with the DT = AI < AT outcome. Following Ryan's procedure, the corresponding critical levels for those four results are adjusted to compute the overall critical level associated with this conclusion.

Thus, the statistical conclusions are in one-to-one correspondence with the research hypotheses and provide concise answers on a "per aspect" basis to the questions of interest. Further details and complete listing of the statistical conclusions for this study are presented in Section IV, Results.

Step 11–Research Interpretations: The final step in the approach is to interpret the statistical conclusions in view of any remaining research framework(s). These research interpretations provide the opportunity to augment the objective findings of the study with the researcher's own subjective judgments and interpretations. The second research framework mentioned above, namely, the general beliefs governing the expected outcomes for the entire set of programming aspects, was considered important. However, this particular research framework can only be utilized for research interpretations, since it is not amenable to rigorous manipulation. Nonetheless, within this framework which is based upon intuitive understanding about the software development environments under consideration, the study bears its most interesting results and implications. Further details and discussion of the research interpretations of this study appear in Section V, Interpretation.

IV. Results

The immediate results of the study are the statistical conclusions inferred from the experiment for each programming aspect considered. They state 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). Each statistical conclusion is expressed in the concise form of a three-way comparison outcome "equation." The equality AI = AT = DT expresses the null conclusion that there is no systematic difference among the groups. An inequality, e.g., AT < (ai) < DT, AI < AT = DT, or DT < AI < AT, expresses a nonnull (or alternative) conclusion that there are certain systematic difference(s) among the groups in stated direction(s). A critical level (or risk) value is also associated with each nonnull (or alternative) conclusion, 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.

Table I gives the complete set of statistical conclusions, arranged by programming aspect. Instances of nonnull (or alternative) conclusions, indicating some distinction among the groups on the basis of a particular programming aspect, are itemized in English prose form at the end of this section.

Examination of the table immediately indicates that roughly half of the programming aspects (particularly product aspects), which were all expected *a priori* to show some distinction among the groups, failed in actuality to do so. However, several of the null conclusions may indicate characteristics

inherent to the application itself. As one example, the basic symbol-table/scanner/parser/code-generator nature of a compiler strongly influences the way the system is modularized and thus practically determines the number of modules in the final product (give or take some occasional slight variation due to other design decisions).

Impact Evaluation

These statistical conclusions have a certain objective character—since they are statistically inferred from empirical data—and their collective impact may be objectively evaluated according to the following statistical principle [27, p. 84-85]. Whenever a series of statistical tests (or experiments) are made, all at a fixed level of significance (for example, 0.10), a corresponding percentage (in the example, 10 percent) of the tests are expected *a priori* to reject the null hypothesis in the complete absence of any true effect (i.e., due to chance alone). This expected rejection percentage provides a comparative index of the true impact of the test results as a whole (in the example, a 25 percent actual rejection percentage would indicate that a truely significant effect, other than chance alone, was operative).

The details of this impact evaluation for the study's objective results, broken down into appropriate categories, are presented in the following table. The evaluation was performed at the $\alpha = 0.20$ significance level used for screening purposes, hence the expected rejection percentage for any category was 20 percent. For each category of aspects, the table gives the number of programming aspects, the expected (rounded to whole numbers) and actual numbers of rejections (of the null conclusion in favor of a directional alternative), and the expected and actual rejection percentages. Strong statistical impact is demonstrated by an actual rejection percentage well above the expected rejection percentage.

category	number of aspects	expected number of rejections	actual number of rejections	expected rejection percentage	actual rejection percentage
"confirmatory" aspects	35	7	19	20.0	54.3
process aspects only	6	1	6	20.0	100.0
product aspects only	29	6	13	20.0	44.8

The table shows that the results do have strong statistical impact. On the whole, process aspects have more impact than product aspects, but all of the observed quantitative distinctions among the three groups bear statistical impact. They are better explained as consequences of some true effect related to the experimental treatments, rather than as random phenomena.

Individual Highlights

The purpose of this subsection is simply to highlight the individual differences observed in the study, by itemizing the nonnull conclusions in English.

1) According to the DT < AI = AT outcome on the COMPUTER JOB STEPS aspect, the disciplined teams used very noticeably fewer computer job steps (i.e., module compilations, program executions, and miscellaneous job steps) than both the ad hoc individuals and the ad hoc teams. As metrics, this aspect and its subclassifications directly represent machine costs, in units of basic computer system operations, and indirectly reflect human costs, since each operation necessitates a certain expenditure of programmer time/effort.

2) This same difference was 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 DT < AT = AI outcomes on the COMPUTER JOB STEPS\MODULE COMPILATION, COMPUTER JOB STEPS\MODULE COMPILATION\UNIQUE, COMPUTER JOB STEPS\PROGRAM EXECUTION, and COMPUTER JOB STEPS/ESSENTIAL/ aspects, respectively.

3) According to the DT < AI = AT outcome on the PROGRAM CHANGES aspect [13] the disciplined teams required very noticeably fewer textural revisions to build and debug the software than the ad hoc individuals and the ad hoc teams. As a metric, this aspect has been shown to correlate well with total number of error occurrences determined via human inspection.

4) There was a definite trend for the ad hoc individuals and disciplined teams to have produced fewer total symbolic lines (including comments, compiler directives, statements, declarations, etc.) than the ad hoc teams, according to the DT = AI < AT outcome on the LINES aspect. There is evidence, as indicated by the lower critical level, of a stronger pairwise difference between ad hoc individuals and ad hoc teams than between disciplined teams and ad hoc teams. This aspect measures the size of the software product.

5) According to the AI < AT = DT outcome on the SEGMENTS aspect, the ad hoc individuals organized their software into noticeably fewer routines (i.e., functions or procedures) than either the ad hoc teams or the disciplined teams. In addition to measuring the size of the software product, this aspect reflects its modularity.

6) The ad hoc individuals displayed a trend toward having a greater number of executable statements per routine than did the ad hoc teams, according to the AT < (dt) < AI outcome on the AVERAGE STATEMENTS PER SEGMENT aspect. As a metric, this aspect represents the length of a typical routine in the delivered source code.

7) According to the DT = AI < AT outcomes on the STATEMENT TYPE COUNTS\IF and STATEMENT TYPE PERCENTAGE\IF aspects, both the ad hoc individuals and the disci-

TABLE I
PROGRAMMING ASPECTS AND STATISTICAL CONCLUSIONS

N.B.: The parenthesized numbers to the right refer to the explanatory notes in Appendix I. A simple pair of equal signs (= =) appears in place of the null outcome AI = AT = DT to avoid unnecessary clutter.

Programming Aspect	Location Comparison Outcome	Critical Level	
Development Process Aspects			
COMPUTER JOB STEPS	DT < AI = AT	0.006	(1)
	DT <(at)< AI	0.006	
	DT <(ai)< AT	0.003	
MODULE COMPILATION	DT < AI = AT	0.042	(2)
	DT <(at)< AI	0.042	
	DT <(ai)< AT	0.005	
UNIQUE	DT < AI = AT	0.020	(3)
	DT <(at)< AI	0.020	
	DT <(ai)< AT	0.004	
PROGRAM EXECUTION	DT < AI = AT	0.041	(4)
	DT <(at)< AI	0.011	
	DT <(ai)< AT	0.041	
ESSENTIAL	DT < AI = AT	0.007	(5)
	DT <(at)< AI	0.003	
	DT <(ai)< AT	0.007	
PROGRAM CHANGES	DT < AI = AT	0.006	(6)
	DT <(at)< AI	0.006	
	DT <(ai)< AT	0.003	
Final Product Aspects			
MODULES	= =		(7)
SEGMENTS	AI <AT= DT	0.109	(8)
	AI <(dt)< AT	0.077	
	AI <(at)< DT	0.109	
LINES	DT = AI< AT	0.109	(9)
	AI <(dt)< AT	0.045	
	DT <(ai)< AT	0.109	
STATEMENTS	= =		(10)

Programming Aspect	Location Comparison Outcome	Critical Level	
Final Product Aspects			
STATEMENT TYPE COUNTS :			(11)
IF	DT = AI< AT	0.139	(12)
	AI <(dt)< AT	0.139	
	DT <(ai)< AT	0.066	
CASE	= =		(13)
WHILE	= =		(14)
EXIT	= =		(15)
RETURN	DT = AI< AT	0.159	(16)
	AI <(dt)< AT	0.159	
	DT <(ai)< AT	0.047	
STATEMENT TYPE PERCENTAGES :			(11)
IF	DT = AI< AT	0.197	(12)
	AI <(dt)< AT	0.197	
	DT <(ai)< AT	0.054	
CASE	AT <(ai)< DT	0.157	(13)
WHILE	= =		(14)

TABLE I (CONTINUED)

Programming Aspect	Location Comparison Outcome	Critical Level	
Final Product Aspects			
EXIT	= =		(15)
RETURN	= =		(16)
AVERAGE STATEMENTS PER SEGMENT	AT <(dt)< AI	0.139	(17)
AVERAGE STATEMENT NESTING LEVEL	= =		(18)
DECISIONS	DT <(ai)< AT	0.090	(19)
TOKENS	= =		(20)
AVERAGE TOKENS PER STATEMENT	= =		(20)
DATA VARIABLE SCOPE COUNTS :			(21)
GLOBAL	AI <(dt)< AT	0.085	(22)
PARAMETER	AI <(dt)< AT	0.123	(22)
LOCAL	= =		(22)
DATA VARIABLE SCOPE PERCENTAGES :			(21)
GLOBAL	= =		(22)
PARAMETER	AI <(dt)< AT	0.111	(22)
LOCAL	AT =DT< AI AT <(dt)< AI DT <(at)< AI	0.197 0.197 0.066	(22)
(SEGMENT, GLOBAL) USAGE PAIR RELATIVE PERCENTAGE			(23)
(SEGMENT, GLOBAL, SEGMENT) DATA BINDINGS :			(24)
ACTUAL			(25)
POSSIBLE	AI <(dt)< AT	0.197	(25)
RELATIVE PERCENTAGE			(25)

plined teams coded noticeably fewer IF statements than the ad hoc teams, in terms of both total number and percentage of total statements. In both cases, it should be noted that the more significant pairwise difference lies between disciplined teams and ad hoc teams. These aspects are two of the earliest proposed and more commonly accepted measures of program complexity.

8) According to the DT < (ai) < AT outcome on the DECISIONS aspect, the disciplined teams tended to code fewer decisions (i.e., IF, WHILE, or CASE statements) than the ad hoc teams. As a metric, this aspect represents control flow complexity; it is closely associated with a recently proposed graph theoretic complexity measure [19].

9) The disciplined teams and the ad hoc individuals both coded fewer RETURN statements than the ad hoc teams, according to the DT = AI < AT outcome on the STATEMENT TYPE COUNTS\RETURN aspect, with the stronger pairwise difference separating disciplined teams and ad hoc teams. This aspect reflects a degree of deviation from rigorously structured code.

10) The disciplined teams coded a higher percentage of CASE statements than the ad hoc teams, according to the AT < (ai) < DT outcome on the STATEMENT TYPE PERCENTAGES\CASE aspect. This aspect reflects the organization of low-level tests into a more concise control structure.

11) The ad hoc individuals tended to use fewer global variables than the ad hoc teams, according to the AI < (dt) < AT outcome on the DATA VARIABLE SCOPE COUNTS\GLOBAL aspect. As metrics, this aspect and the others dealing with scope reflect the organization and accessibility of data within a program.

12) The ad hoc individuals also tended to use fewer parameter variables than the ad hoc teams, in terms of both total number and percentage of declared data variables, according to the AI < (dt) < AT outcomes on the DATA VARIABLE SCOPE COUNTS\PARAMETER and DATA VARIABLE SCOPE PERCENTAGES\PARAMETER aspects.

13) According to the AT = DT < AI outcome on the DATA VARIABLE SCOPE PERCENTAGES\LOCAL aspect, the ad hoc individuals had a larger percentage of local variables compared to the total number of declared data variables than either the ad hoc teams or the disciplined teams. The stronger pairwise differentiation lies between disciplined teams and ad hoc individuals.

14) There was a slight trend for the ad hoc individuals to have fewer potential data bindings [26] (i.e., occurrences of the situation where a global variable could be modified by one segment and accessed by another due to the software's modularization) than the ad hoc teams, according to the AI < (dt) < AT outcome on the (SEG, GLOBAL, SEG) DATA BINDINGS\POSSIBLE aspect. As a metric, this aspect represents the potential number of unique communication paths via globals between pairs of segments.

V. INTERPRETATION

The study's derived results, called research interpretations, consist of an evaluation of the statistical conclusions presented in Section IV, based upon a set of general beliefs regarding software development. These beliefs were formulated by the researchers prior to conducting the experiment. Pertaining to both the process and product of software development, the beliefs are

(B1) that methodological discipline is a key influence on the general efficiency of the software process;

(B2) that the disciplined methodology reduces the cost and complication of the process;

(B3) that the preferred direction of differences on process aspects is clear and undebatable, due to the tangibleness of the process aspects themselves and the direct applicability of expected values in terms of average cost estimates;

(B4) that "mental cohesiveness" (or conceptual integrity [9, pp. 41-50]) is a key influence on the general quality of the software product;

(B5) that a programming team is naturally burdened (relative to an individual programmer) by the organizational overhead and risk of error-prone misunderstanding inherent in coordinating and interfacing the thoughts and efforts of those on the team;

(B6) that the disciplined methodology induces an effective mental cohesiveness, enabling a programming team to behave more like an individual programmer with respect to conceptual control over the program, its design, its structure, etc., because of the discipline's antiregressive, complexity-controlling effects that compensate for the inherent organization overhead of a team; and

(B7) that the preferred direction of differences on product aspects is not always clear (occasionally even subject to diverging viewpoints), due to the intangibleness of many of the product aspects.

In relation to these general beliefs, each possible comparison outcome acquires additional meaning, either substantiating or contravening some subset of the beliefs. For process aspects and beliefs (B1)-(B3)

a) the level-2 outcome DT < AI = AT is directly supportive of these beliefs;

b) the level-3 outcomes DT < AI < AT and DT < AT < AI and the level-1 outcomes DT < (ai) < AT and DT < (at) < AI are indirectly supportive of these beliefs;

c) the level-0 outcome AI = AT = DT may discredit these beliefs, or it may be considered neutral for anyone of several possible reasons [1) the critical level for a nonnull outcome is just not low enough, so the aspect defaults to the null outcome; 2) the aspect simply reflects something characteristic of the application itself (or another factor common to all the groups in the experiment); or 3) the aspect actually measures something fundamental to software development phenomena in general and would always result in the null outcome]; and

d) all other outcomes discredit these beliefs.

For product aspects and beliefs (B4)-(B7)

a) the level-2 outcome AT ≠ DT = AI, which is equivalent to AT < DT = AI or DT = AI < AT, is directly supportive of these beliefs;

b1) the level-3 outcomes AI < DT < AT and AT < DT < AI may be considered as approximations to the "preferred" level-2 outcome in a) above [DT is distinct from AT but falls short of AI, due to lack of experience or maturity in the disciplined methodology.];

b2) the level-1 outcomes AT ≠ DT and AI ≠ AT may also

be considered as approximations to the "preferred" level-2 outcome in a) above [AI ≠ AT, which is equivalent to AI < (dt) < AT or AT < (dt) < AI, supports the beliefs (B4), (B5) that mental cohesiveness influences the quality of a product and that an ad hoc team is burdened by its organizational overhead. DT ≠ AT, which is equivalent to DT < (ai) < AT or AT < (ai) < DT, supports the belief (B6) that the disciplined methodology affects the behavior of a team.];

c) the level-0 outcome AI = AT = DT may discredit these beliefs, or it may be considered neutral for anyone of several possible reasons [as given in c) above]; and

d) all other outcomes discredit one or more of these beliefs.

The study's interpretation therefore consists of a general assessment of how well the research conclusions have borne out the general beliefs. On the whole, the study's findings do support the general beliefs presented above, although a few conclusions exist which are inconsistent with them.

Overwhelming support comes in the category of comparisons on process aspects, in which the research conclusions are distinguished by their low critical levels and by their unanimous DT < AI = AT outcome. Fairly strong support also comes in the category of comparisons on product aspects, for which the only negative evidence (besides the neutral AI = AT = DT outcomes) appeared in the form of two AI ≠ AT = DT outcomes. These indicate some areas in which the disciplined methodology was apparently ineffective in modifying a team's behavior toward that of an individual, possibly due to a lack of fully developed training/experience with the methodology.

Thus, according to this interpretation, the study's findings strongly substantiate the claims

(C1) that methodological discipline is a key influence on the general efficiency of the software development process, and

(C2) that the disciplined methodology significantly reduces the material costs of software development.

The claims

(C3) that mental cohesiveness is a key influence on the general quality of the software development product,

(C4) that, relative to an individual programmer, an ad hoc programming team is mentally burdened by its organizational overhead, and

(C5) that the disciplined methodology offsets the mental burden of organizational overhead and enables a disciplined programming team to behave more like an individual programmer relative to the developed software product

are moderately substantiated by the study's findings.

It should be noted that there is a simpler (albeit weaker) interpretive model that covers all of the experimental results. With the beliefs that a disciplined methodology provides for the minimum process cost and results in a product which in some aspects approximates the product of an individual and at worst approximates the product developed by an ad hoc team, the suppositions are DT < AI *and* DT < AT with respect to process and AI ≤ DT ≤ AT *or* AT ≤ DT ≤ AI with respect to product. The study's statistical conclusions fit this model without exception.

The interpretations presented here are neither exhaustive nor unique. They express the researchers' own estimation of the study's implications and general import, according to their professional intuitions about programming and software. It is anticipated that the reader and other researchers might formulate additional or alternative interpretations of the study's empirical results, using their own intuitive judgments. Other interpretations may be found in [5], [22].

VI. Conclusion

A practical methodology was designed and developed for experimentally and quantitatively investigating software development phenomena. 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 structured 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. It must be remembered, however, that the results and interpretation of this study are derived from a limited subject population and a set of measures assumed to be associated with software cost and quality. Further studies replicating these experiments in other environments should be performed.

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. These measures are assumed to be associated with process aspects considered in the study, namely, PROGRAM CHANGES and COMPUTER JOB STEPS/ESSENTIAL, respectively. The statistical conclusions for both these aspects affirmed DT < AI = AT outcomes at very low (<0.01) significance levels, indicating that on the average the disciplined programming teams "scored" lower than either the ad hoc individual programmers or the ad hoc programming teams, which both "scored" about the same. Thus, the evidence collected in this study 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 of this study's 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. The superficiality of many of the product measures, together with the small sample sizes, may be largely 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 evident.

It is important that quantitative evidence be gathered to evaluate software methods and tools. The results of these experiments are being 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 [8]. This type of research is being pursued [3], [4], extending the study to include more sophisticated and promising aspects, such as Halstead's software science quantities [14] and other software complexity metrics [19].

APPENDIX I
EXPLANATORY NOTES FOR THE PROGRAMMING ASPECTS

The following numbered paragraphs, keyed to the list of aspects in Table I and in Appendix II, describe each of the programming aspects considered in the study. Various system- or language-dependent terms (e.g., module, segment) are also defined here.

(1) A *computer job step* is a single indivisible activity performed on a computer at the operating system command level which is nonincidental to the development effort and involves a nontrivial expenditure of computer or human resources. 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. Only compilations of modules comprising the final software product (or logical predecessors thereof) are counted as COMPUTER JOB STEPS\MODULE COMPILATION.

(3) A *unique* module compilation is one in which the source code compiled is textually distinct from that of any 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) 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).

(6) The *program changes* metric [13] 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.

(7) 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.

(8) A *segment* 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 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); 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.

(9) The LINES aspect counts every textual line in the source code of the complete program, including comments, compiler directives, variable declarations, executable statements, etc.

(10) The STATEMENTS aspect counts only the executable constructs in the source code of the complete program. 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.

(11) The group of aspects named STATEMENT TYPE COUNTS, etc., gives the absolute number of executable statements of certain types. The group of aspects named STATEMENT TYPE PERCENTAGES, etc., gives the relative percentage of certain types of statements, compared with the total number of executable statements.

(12) Both ifthen and ifthenelse constructs are counted as IF statements.

(13) The CASE statement provides for selection from several alternatives, depending upon the value of an expression. A case construct with n alternatives is logically and semantically equivalent to a certain pattern of n nested ifthenelse constructs.

(14) The WHILE statement is the only iteration or looping construct provided by the implementation language SIMPL-T.

(15) 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.

(16) 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.

(17) 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.

(18) 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, starting at 1 for a statement at the outermost level of each segment and increasing by 1 for successively nested statements.

(19) The DECISIONS aspect simply counts the total number of IF, CASE, and WHILE statements within the complete source code.

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

(21) A *data variable* is an individually named scalar or array of scalars. In the implementation language SIMPL-T, there are three data *types* for scalars: integer, character, and (varying length) string; there is one kind of data *structure* (besides scalar): single dimensional array, with zero-origin subscript range; and there are several levels of *scope* for data variables (as explained in note (22) below). In addition, all data variables in a SIMPL-T program must be explicitly declared, with attributes fully specified. The total number of data variables includes each data variable declared in the complete program once, regardless of its type, structure, or scope. Note that each array is counted as a single data variable.

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.

(22) In the implementation language SIMPL-T, data variables can have any one of three levels of *scope*–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. *Global* variables are accessible by name to each of the segments in the module(s) in which they are declared, and their values are usually manipulated by several segments. Formal *parameters* are accessible by name only within the enclosing (called) segment, but their values are not completely unrelated to the calling segment (since parameters are passed either by value or by reference). *Locals* are accessible by name only within the enclosing segment, and their values are completely isolated from any other segment.

(23) A segment-global *usage pair* (p, r) is 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.

The *actual* usage pair count is the absolute number of true usage pairs (p, r): the global variable r is actually used by segment p. The *possible* usage pair count is the absolute number of potential usage pairs (p, r), given the program's global variables and their declared scope: if the scope of global variable r contains segment p, then p could potentially modify or access r. The count of possible usage pairs is computed as the sum of the number of segments in each global variable's scope. The (SEG, GLOBAL) USAGE RELATIVE PERCENTAGE count is a way of normalizing the number of usage pairs since it is simply the ratio (expressed as a percentage) of actual usage pairs to possible usage pairs.

(24) A segment-global-segment *data binding* (p, r, q) [26] is an occurrence of the following arrangement in a program: 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 binding (p, r, p) is different from the binding (q, r, p) which may also exist; occurrences such as (p, r, q) are not counted as data bindings.

(25) 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. 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. 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.

APPENDIX II

RAW DATA FOR THE PROGRAMMING ASPECTS

For each measured programming aspect considered in the study and reported in this paper, the observed raw data scores are listed below in ascending order and identified both as to the type of programming environment–ad hoc individuals (AI), ad hoc teams (AT), or disciplined teams (DT)–and as to the particular numbered subject (an individual or a team) within that environment. For example, "AT(4)" identifies the fourth ad hoc team participating in the experiment.

N.B.: The parenthesized numbers to the right of the programming aspect labels refer to the explanatory notes in Appendix I.

COMPUTER JOB STEPS (1)

DT(2) =	44
DT(6) =	58
DT(1) =	67
DT(3) =	68
DT(4) =	79
AI (6) =	87
DT(5) =	90
DT(7) =	123
AT(5) =	150
AI (3) =	151
AI (1) =	159
AT(6) =	164
AT(4) =	173
AI (5) =	176
AI (4) =	183
AT(1) =	216
AT(3) =	266
AI (2) =	357
AT(2) =	372

COMPUTER JOB STEPS \ MODULE COMPILATION (1), (2)

DT(2) =	32
AI (6) =	34
DT(1) =	34
DT(6) =	38
DT(5) =	49
DT(3) =	51
DT(4) =	52
DT(7) =	70
AT(4) =	74
AI (1) =	83
AI (3) =	87
AT(5) =	104
AI (5) =	110
AI (4) =	123
AT(6) =	133
AT(1) =	147
AT(3) =	162
AI (2) =	176
AT(2) =	199

COMPUTER JOB STEPS \ MODULE COMPILATION \ UNIQUE (1), (2), (3)

DT(2) =	25
DT(1) =	27
DT(3) =	30
AI (6) =	31
DT(5) =	33
DT(6) =	35
DT(4) =	42
DT(7) =	59
AT(4) =	62
AI (4) =	70
AT(6) =	73
AI (1) =	79
AI (3) =	79
AT(5) =	98
AI (5) =	100
AT(1) =	118
AI (2) =	129
AT(3) =	140
AT(2) =	159

COMPUTER JOB STEPS \ PROGRAM EXECUTION (1), (4)

DT(2) =	12
DT(3) =	16
DT(6) =	20
DT(4) =	23
AT(6) =	29
DT(1) =	33
DT(5) =	39
AT(5) =	42
AI (3) =	49
AI (6) =	52
AI (4) =	53
DT(7) =	53
AI (5) =	63
AT(1) =	64
AI (1) =	76
AT(3) =	90
AT(4) =	96
AI (2) =	163
AT(2) =	173

COMPUTER JOB STEPS \ ESSENTIAL (1), (5)

DT(2) =	37
DT(3) =	46
DT(6) =	55
DT(1) =	60
DT(4) =	65
DT(5) =	72
AI (6) =	83
AT(6) =	102
DT(7) =	112
AI (4) =	123
AI (3) =	128
AT(5) =	140
AI (1) =	155
AT(4) =	158
AI (5) =	163
AT(1) =	182
AT(3) =	230
AI (2) =	292
AT(2) =	332

PROGRAM CHANGES (6)

DT(4) =	111
DT(7) =	114
DT(2) =	120
DT(3) =	136
DT(6) =	159
AI (6) =	187
DT(1) =	223
DT(5) =	251
AI (3) =	270
AI (2) =	281
AT(6) =	287
AT(1) =	301
AI (4) =	316
AT(4) =	394
AT(5) =	493
AI (5) =	525
AI (1) =	539
AT(3) =	554
AT(2) =	1107

MODULES (7)

AT(1) =	1
AT(2) =	1
AI (1) =	2
AI (5) =	2
AI (6) =	2
AT(4) =	2
DT(1) =	2
AI (2) =	3
DT(2) =	3
DT(5) =	3
DT(7) =	3
AI (4) =	4
AT(3) =	4
DT(6) =	5
DT(4) =	6
DT(3) =	8
AT(5) =	9
AI (3) =	10
AT(6) =	15

SEGMENTS (8)

AI (2) =	21
AI (1) =	24
AI (6) =	25
AI (5) =	33
DT(2) =	33
DT(6) =	33
AI (3) =	34
AT(2) =	38
DT(3) =	38
AT(3) =	39
AT(6) =	42
DT(4) =	42
DT(7) =	42
AT(1) =	45
AI (4) =	47
AT(4) =	48
DT(1) =	52
DT(5) =	52
AT(5) =	74

LINES (9)

AI (6) =	579
AI (1) =	836
DT(2) =	894
AI (2) =	944
DT(3) =	1083
AI (5) =	1087
AT(1) =	1138
AI (4) =	1155
DT(7) =	1235
DT(4) =	1267
DT(5) =	1269
AT(3) =	1394
AI (3) =	1559
DT(1) =	1579
AT(2) =	1588
DT(6) =	1600
AT(6) =	1675
AT(5) =	2078
AT(4) =	2186

STATEMENTS (10)

AI (6) =	378
AI (1) =	432
DT(3) =	456
DT(7) =	499
DT(2) =	502
AI (2) =	556
AT(4) =	590
DT(4) =	617
AI (5) =	629
AT(1) =	631
DT(5) =	640
DT(6) =	643
AI (4) =	647
AT(2) =	654
AT(6) =	681
AT(3) =	691
AI (3) =	738
AT(5) =	798
DT(1) =	800

STATEMENT TYPE COUNTS \ IF (11), (12)

AI (6) =	27
DT(7) =	38
AI (2) =	43
DT(3) =	44
AI (1) =	49
DT(2) =	62
DT(4) =	63
AT(4) =	78
AI (4) =	80
DT(1) =	83
AT(1) =	88
DT(5) =	89
DT(6) =	90
AT(3) =	97
AI (5) =	100
AI (3) =	110
AT(5) =	114
AT(2) =	116
AT(6) =	124

STATEMENT TYPE COUNTS \ CASE (11), (13)

AI (5) =	1
AT(1) =	1
AT(2) =	4
AT(6) =	4
DT(2) =	4
DT(3) =	4
DT(7) =	4
AI (3) =	6
AI (6) =	6
AT(4) =	6
AT(5) =	6
AI (1) =	7
DT(4) =	7
DT(6) =	7
AT(3) =	10
AI (2) =	11
AI (4) =	11
DT(5) =	12
DT(1) =	14

STATEMENT TYPE COUNTS \ WHILE (11), (14)

DT(4) =	17
AI (6) =	18
AI (1) =	19
AI (5) =	21
AT(4) =	21
DT(6) =	21
DT(3) =	22
DT(5) =	22
AT(2) =	24
AT(6) =	24
DT(2) =	24
DT(7) =	25
AT(5) =	28
AI (2) =	29
AI (4) =	30
AT(1) =	31
AI (3) =	34
DT(1) =	34
AT(3) =	35

STATEMENT TYPE COUNTS \ EXIT (11), (15)

AI (6) =	0
AT(1) =	0
AT(2) =	0
AT(3) =	0
AT(4) =	0
DT(1) =	0
DT(2) =	0
DT(3) =	0
DT(4) =	0
DT(5) =	0
AI (1) =	1
AI (2) =	1
DT(7) =	2
AI (4) =	3
DT(6) =	3
AT(6) =	6
AI (5) =	8
AT(5) =	13
AI (3) =	15

STATEMENT TYPE COUNTS \
RETURN
(11), (16)

AI (6) =	36
AI (2) =	47
AI (3) =	47
DT(2) =	47
DT(3) =	47
DT(4) =	48
DT(6) =	48
AT(4) =	50
DT(7) =	50
AI (1) =	53
AT(2) =	53
DT(1) =	54
AI (5) =	59
AI (4) =	60
AT(3) =	64
DT(5) =	65
AT(1) =	99
AT(6) =	109
AT(5) =	118

STATEMENT TYPE PERCENTAGES \
IF
(11), (12)

AI (6) =	7.1
DT(7) =	7.6
AI (2) =	7.7
DT(3) =	9.6
DT(4) =	10.2
DT(1) =	10.4
AI (1) =	11.3
AI (4) =	12.4
DT(2) =	12.4
AT(4) =	13.2
AT(1) =	13.9
DT(5) =	13.9
AT(3) =	14.0
DT(6) =	14.0
AT(5) =	14.3
AI (3) =	14.9
AI (5) =	15.9
AT(2) =	17.7
AT(6) =	18.2

STATEMENT TYPE PERCENTAGES \
CASE
(11), (13)

AI (5) =	0.2
AT(1) =	0.2
AT(2) =	0.6
AT(6) =	0.6
AI (3) =	0.8
AT(5) =	0.8
DT(2) =	0.8
DT(7) =	0.8
DT(3) =	0.9
AT(4) =	1.0
DT(4) =	1.1
DT(6) =	1.1
AT(3) =	1.4
AI (1) =	1.6
AI (6) =	1.6
AI (4) =	1.7
DT(1) =	1.8
DT(5) =	1.9
AI (2) =	2.0

STATEMENT TYPE PERCENTAGES \
WHILE
(11), (14)

DT(4) =	2.8
AI (5) =	3.3
DT(6) =	3.3
DT(5) =	3.4
AT(5) =	3.5
AT(6) =	3.5
AT(4) =	3.6
AT(2) =	3.7
DT(1) =	4.3
AI (1) =	4.4
AI (3) =	4.6
AI (4) =	4.6
AI (6) =	4.8
DT(2) =	4.8
DT(3) =	4.8
AT(1) =	4.9
DT(7) =	5.0
AT(3) =	5.1
AT(2) =	5.2

STATEMENT TYPE PERCENTAGES \
EXIT
(11), (15)

AI (6) =	0.0
AT(1) =	0.0
AT(2) =	0.0
AT(3) =	0.0
AT(4) =	0.0
DT(1) =	0.0
DT(2) =	0.0
DT(3) =	0.0
DT(4) =	0.0
DT(5) =	0.0
AI (1) =	0.2
AI (2) =	0.2
DT(7) =	0.4
AI (4) =	0.5
DT(6) =	0.5
AT(6) =	0.9
AI (5) =	1.3
AT(5) =	1.6
AI (3) =	2.0

STATEMENT TYPE PERCENTAGES \
RETURN
(11), (16)

AI (3) =	6.4
DT(1) =	6.8
DT(6) =	7.5
DT(4) =	7.8
AT(2) =	8.1
AI (2) =	8.5
AT(4) =	8.5
AI (4) =	9.3
AT(3) =	9.3
AI (5) =	9.4
DT(2) =	9.4
AI (6) =	9.5
DT(7) =	10.0
DT(5) =	10.2
DT(3) =	10.3
AI (1) =	12.3
AT(5) =	14.8
AT(1) =	15.7
AT(6) =	16.0

AVERAGE STATEMENTS PER SEGMENT (17)

AT(5) =	10.8
DT(7) =	11.9
DT(3) =	12.0
AT(4) =	12.3
DT(5) =	12.3
AI (4) =	13.8
AT(1) =	14.0
DT(4) =	14.7
AI (6) =	15.1
DT(2) =	15.2
DT(1) =	15.4
AT(6) =	16.2
AT(2) =	17.2
AT(3) =	17.7
AI (1) =	18.0
AI (5) =	19.1
DT(6) =	19.5
AI (3) =	21.7
AI (2) =	26.5

AVERAGE STATEMENT NESTING LEVEL (18)

AT(1) =	1.9
AT(5) =	1.9
AT(4) =	2.0
DT(2) =	2.0
DT(3) =	2.0
DT(7) =	2.0
AI (6) =	2.1
DT(4) =	2.1
AI (4) =	2.2
DT(5) =	2.2
AI (5) =	2.3
AT(2) =	2.3
AT(3) =	2.3
DT(1) =	2.3
AI (1) =	2.4
AI (2) =	2.4
DT(6) =	2.4
AI (3) =	2.6
AT(6) =	2.7

DECISIONS (19)

AI (6) =	51
DT(7) =	67
DT(3) =	70
AI (1) =	75
AI (2) =	83
DT(4) =	87
DT(2) =	90
AT(4) =	105
DT(6) =	118
AT(1) =	120
AI (4) =	121
AI (5) =	122
DT(5) =	123
DT(1) =	131
AT(3) =	142
AT(2) =	144
AT(5) =	148
AI (3) =	150
AT(6) =	152

TOKENS (20)

AI (6) =	1878
DT(7) =	2113
DT(3) =	2268
AI (1) =	2313
DT(2) =	2348
AT(4) =	2976
AI (5) =	3270
AI (2) =	3277
AT(6) =	3508
AT(1) =	3622
AT(2) =	3669
DT(5) =	3777
AI (4) =	3792
DT(6) =	3792
AI (3) =	3907
DT(4) =	4016
AT(5) =	4198
AT(3) =	4269
DT(1) =	4471

AVERAGE TOKENS PER STATEMENT (20)

DT(7) =	4.2
DT(2) =	4.7
AI (6) =	5.0
AT(4) =	5.0
DT(3) =	5.0
AI (5) =	5.2
AT(6) =	5.2
AI (3) =	5.3
AT(5) =	5.3
AI (1) =	5.4
AT(2) =	5.6
DT(1) =	5.6
AT(1) =	5.7
AI (2) =	5.9
AI (4) =	5.9
DT(5) =	5.9
DT(6) =	5.9
AT(3) =	6.2
DT(4) =	6.5

DATA VARIABLE SCOPE COUNTS \ GLOBAL (21), (22)

AI (6) =	15
DT(3) =	21
AI (2) =	23
AI (5) =	23
AT(2) =	24
DT(5) =	24
DT(1) =	26
AI (1) =	28
AI (3) =	29
AI (4) =	30
AT(4) =	32
DT(7) =	33
AT(6) =	35
AT(5) =	37
AT(3) =	38
DT(6) =	38
AT(1) =	46
DT(4) =	86
DT(2) =	91

DATA VARIABLE SCOPE COUNTS \ PARAMETER (21), (22)

AI (5) =	4
AI (6) =	4
DT(2) =	6
DT(7) =	8
AI (1) =	10
AI (2) =	11
AT(6) =	13
AI (3) =	15
AT(2) =	20
DT(6) =	24
DT(3) =	26
AT(1) =	31
AT(4) =	33
AI (4) =	34
AT(3) =	38
AT(5) =	41
DT(5) =	51
DT(1) =	54
DT(4) =	54

DATA VARIABLE SCOPE COUNTS \ LOCAL (21), (22)

AI (5) =	16
AI (6) =	19
AI (1) =	23
DT(2) =	23
DT(3) =	25
DT(5) =	25
DT(7) =	28
AT(3) =	30
AT(6) =	31
AI (3) =	34
DT(4) =	34
AT(4) =	36
DT(6) =	37
AI (2) =	42
AT(2) =	42
AT(5) =	45
AI (4) =	49
AT(1) =	50
DT(1) =	53

DATA VARIABLE SCOPE PERCENTAGES \ GLOBAL (21), (22)

DT(1) =	19.5
DT(5) =	24.0
AI (4) =	26.5
AT(2) =	27.9
DT(3) =	29.2
AT(5) =	30.1
AI (2) =	30.3
AT(4) =	31.7
AT(3) =	35.8
AT(1) =	36.2
AI (3) =	37.2
DT(6) =	38.4
AI (6) =	39.5
AT(6) =	44.3
AI (1) =	45.9
DT(7) =	47.8
DT(4) =	49.4
AI (5) =	53.5
DT(2) =	75.8

DATA VARIABLE SCOPE PERCENTAGES \ PARAMETER (21), (22)

DT(2) =	5.0
AI (5) =	9.3
AI (6) =	10.5
DT(7) =	11.6
AI (2) =	14.5
AI (1) =	16.4
AT(6) =	16.5
AI (3) =	19.2
AT(2) =	23.3
DT(6) =	24.2
AT(1) =	24.4
AI (4) =	30.1
DT(4) =	31.0
AT(4) =	32.7
AT(5) =	33.3
AT(3) =	35.8
DT(3) =	36.1
DT(1) =	40.6
DT(5) =	51.0

DATA VARIABLE SCOPE PERCENTAGES \ LOCAL (21), (22)

DT(2) =	19.2
DT(4) =	19.5
DT(5) =	25.0
AT(3) =	28.3
DT(3) =	34.7
AT(4) =	35.6
AT(5) =	36.6
AI (5) =	37.2
DT(6) =	37.4
AI (1) =	37.7
AT(6) =	39.2
AT(1) =	39.4
DT(1) =	39.8
DT(7) =	40.6
AI (4) =	43.4
AI (3) =	43.6
AT(2) =	48.8
AI (6) =	50.0
AI (2) =	55.3

(SEGMENT, GLOBAL) USAGE PAIR RELATIVE PERCENTAGE (23)

AT(1) =	7.8
AT(5) =	9.6
AT(4) =	11.4
DT(7) =	13.0
AT(2) =	14.7
DT(1) =	15.6
AI (1) =	15.7
DT(2) =	17.6
DT(4) =	18.3
AI (4) =	21.4
DT(5) =	25.0
AI (5) =	25.8
AI (6) =	26.8
AT(3) =	27.2
DT(6) =	27.6
AT(6) =	30.1
AI (3) =	31.5
AI (2) =	37.1
DT(3) =	43.2

(SEGMENT, GLOBAL, SEGMENT) DATA BINDINGS \ ACTUAL (24), (25)

DT(3) =	121
DT(2) =	154
DT(4) =	164
AT(3) =	184
DT(7) =	210
AI (6) =	214
AT(2) =	221
AI (1) =	244
DT(6) =	260
AI (3) =	280
AI (2) =	302
AT(6) =	310
AT(5) =	360
AT(4) =	398
AI (4) =	438
AI (5) =	590
AT(1) =	1087
DT(1) =	1104
DT(5) =	1337

(SEGMENT, GLOBAL, SEGMENT) DATA BINDINGS \ POSSIBLE (24), (25)

DT(3) =	2812
AI (2) =	3588
AT(3) =	5164
AI (6) =	6612
DT(1) =	7166
DT(2) =	7434
DT(4) =	7500
AI (3) =	8922
AT(6) =	8974
AI (1) =	9798
DT(6) =	10834
AI (5) =	15852
DT(5) =	17008
AI (4) =	21309
DT(7) =	31704
AT(2) =	33744
AT(1) =	41500
AT(4) =	49782
AT(5) =	115182

(SEGMENT, GLOBAL, SEGMENT) DATA BINDINGS \ RELATIVE PERCENTAGE (24), (25)

AT(5) =	0.3
AT(2) =	0.7
DT(7) =	0.7
AT(4) =	0.8
AI (4) =	2.1
DT(2) =	2.1
DT(4) =	2.2
DT(6) =	2.4
AI (1) =	2.5
AT(1) =	2.6
AI (3) =	3.1
AI (6) =	3.2
AT(6) =	3.5
AT(3) =	3.6
AI (5) =	3.7
DT(3) =	4.3
DT(5) =	7.9
AI (2) =	8.4
DT(1) =	15.4

Acknowledgment

It is a pleasure to acknowledge colleagues Dr. J. D. Gannon (University of Maryland) and Dr. H. E. Dunsmore (Purdue University) for the constructive criticism and insightful discussion they provided throughout this study. The authors are indebted to Mr. W. D. Brooks (IBM Federal Systems Division) for his technical assistance regarding the statistical data analysis. The authors also thank the referees for their helpful suggestions on improving the presentation of this paper.

References

[1] F. T. Baker, "Structured programming in a production programming environment," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 241–252, June 1975.

[2] V. R. Basili and F. T. Baker, *Tutorial of Structured Programming*, Tutorial from the 11th IEEE Comput. Soc. Conf. (COMPCON 75 Fall), IEEE Cat. 75CH1049-6, revised 1977.

[3] V. R. Basili and D. H. Hutchens, "A study of a family of structural complexity metrics," in *Proc. 19th Annu. ACM/NBS Tech. Symp., Pathways to System Integrity.* Washington, DC, June 1980, pp. 13–15.

[4] V. R. Basili and R. W. Reiter, Jr., "Evaluating automatable measures of software development," in *Proc. IEEE/Poly Workshop on Quantitative Software Models for Reliability, Complexity, and Cost*, Kiameshia Lake, NY, Oct. 1979, IEEE Cat. TH0067-9, pp. 107–116.

[5] —, "An investigation of human factors in software development," *Computer*, vol. 12, pp. 21–38, Dec. 1979.

[6] V. R. Basili and A. J. Turner, "Iterative enhancement: A practical technique for software development," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 390–396, Dec. 1975.

[7] V. R. Basili and A. J. Turner, *SIMPL-T, A Structured Programming Language.* Geneva, IL: Paladin House, 1976.

[8] V. R. Basili and M. V. Zelkowitz, "Analyzing medium-scale software development," in *Proc. 3rd Int. Conf. Software Eng.*, Atlanta, GA, May 1978, IEEE Cat. 78CH1317-7C, pp. 116–123.

[9] F. P. Brooks, Jr., *The Mythical Man-Month.* Reading, MA: Addison-Wesley, 1975.

[10] W. J. Conover, *Practical Nonparametric Statistics.* New York: Wiley, 1971.

[11] O.-J. Dahl, E. W. Dijkstra, and C.A.R. Hoare, *Structured Programming.* New York: Academic, 1972.
[12] E. B. Daley, "Management of software development," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 229–242, May 1977.
[13] H. E. Dunsmore and J. D. Gannon, "Experimental investigation of programming complexity," in *Proc. 16th Annu. ACM/NBS Tech. Symp., Systems and Software.* Washington, DC, June 1977, pp. 117–125.
[14] M. Halstead, *Elements of Software Science.* New York: Elsevier, 1977.
[15] M. A. Jackson, *Principles of Program Design.* New York: Academic, 1975.
[16] R. E. Kirk, *Experimental Design: Procedures for the Behavioral Sciences.* Belmont, CA: Wadsworth, 1968.
[17] R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice.* Reading, MA: Addison-Wesley, 1979.
[18] H. C. Lucas and R. B. Kaplan, "A structured programming experiment," *Comput. J.*, vol. 19, pp. 136–138, May 1976.
[19] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 308–320, Dec. 1976.
[20] G. J. Myers, *Reliable Software through Composite Design.* New York: Petrocelli/Charter, 1975.
[21] G. J. Myers, "A controlled experiment in program testing and code walkthroughs/inspections," *Commun. Ass. Comput. Mach.*, vol. 21, pp. 760–768, Sept. 1978.
[22] R. W. Reiter, Jr., "An experimental investigation of computer program development approaches and computer programming metrics," Ph.D. dissertation (308), Dep. Comput. Sci., Univ. Maryland, Dec. 1979 (forthcoming as Tech. Rep. TR-853).
[23] S. B. Sheppard, B. Curtis, P. Milliman, and T. Love, "Modern coding practices and programmer performance," *Computer*, vol. 12, pp. 41–49, Dec. 1979.
[24] B. Shneiderman, R. Mayer, D. McKay, and P. Heller, "Experimental investigations of the utility of detailed flowcharts in programming," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 373–381, June 1977.
[25] S. Siegel, *Nonparametric Statistics: For the Behavioral Sciences.* New York: McGraw-Hill, 1956.
[26] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115–139, 1974.
[27] J. W. Tukey, "Analyzing data: Sanctification or detective work?," *Amer. Psychol.*, vol. 24, pp. 83–91, Feb. 1969.
[28] N. Wirth, "Program development by stepwise refinement," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 221–227, Apr. 1971.

Victor R. Basili received the Ph.D. degree in computer science from the University of Texas, Austin.

He is currently an Associate Professor of Computer Science at the University of Maryland, College Park, where he has been since 1970. He has been involved in the design and development of several software projects, including the SIMPL family of structured programming languages and is currently involved in the measurement and evaluation of software development at the NASA/Goddard Space Flight Center. His interests lie in software development methodology and the quantitative analysis and evaluation of the software development process and product. This includes such specialized areas as cost modeling, error analysis, and complexity. He has consulted for several government agencies and industrial organizations, including IBM, GE, CSC, NRL, NSWC, and NASA. He has been program chairman for several conferences and has served on several editorial boards.

Dr. Basili is a member of the Association for Computing Machinery and the IEEE Computer Society.

Robert W. Reiter, Jr. (S'79–M'79) was born in Baltimore, MD, on June 7, 1950. He received the S.B. degree in mathematics from the Massachusetts Institute of Technology, Cambridge, in 1972 and the M.S. and Ph.D. degrees in computer science from the University of Maryland, College Park, in 1976 and 1979, respectively.

During the course of his graduate studies in the Department of Computer Science, University of Maryland, he contributed to the enhancement of the SIMPL family of transportable extendable compilers and to the initial formation of the Software Engineering Laboratory at NASA Goddard Space Flight Center, Greenbelt, MD. Since 1980 he has been a staff programmer in the Software Engineering and Technology Department of the IBM Federal Systems Division. His current research interests cover empirical study in software engineering, software development and maintenance methodology, and software metrics.

Dr. Reiter is a member of the Association for Computing Machinery and the IEEE Computer Society.