Cleanroom Software Development: An Empirical Evaluation

RICHARD W. SELBY, MEMBER, IEEE, VICTOR R. BASILI, SENIOR MEMBER, IEEE, AND F. TERRY BAKER

Abstract—**The Cleanroom software development approach is intended to produce highly reliable software by integrating formal methods for specification and design, nonexecution-based program development, and statistically based independent testing. In an empirical study, 15 three-person teams developed versions of the same software system (800–2300 source lines); ten teams applied Cleanroom, while five applied a more traditional approach. This analysis characterizes the effect of Cleanroom on the delivered product, the software development process, and the developers.**

The major results of this study are the following. 1) Most of the developers were able to apply the techniques of Cleanroom effectively (six of the ten Cleanroom teams delivered at least 91 percent of the required system functions). 2) The Cleanroom teams' products met system requirements more completely and had a higher percentage of successful operationally generated test cases. 3) The source code developed using Cleanroom had more comments and less dense control-flow complexity. 4) The more successful Cleanroom developers modified their use of the implementation language; they used more procedure calls and IF statements, used fewer CASE and WHILE statements, and had a lower frequency of variable reuse (average number of occurrences per variable). 5) All ten Cleanroom teams made all of their scheduled intermediate product deliveries, while only two of the five non-Cleanroom teams did. 6) Although 86 percent of the Cleanroom developers indicated that they missed the satisfaction of program execution to some extent, this had no relation to the product quality measures of implementation completeness and successful operational tests. 7) Eighty-one percent of the Cleanroom developers said that they would use the approach again.

Index Terms—**Empirical study, methodology evaluation, off-line software review, software development methodology, software management, software measurement, software testing.**

I. Introduction

THE need for discipline in the software development process and for high quality software motivates the Cleanroom software development approach. In addition to improving the control during development, this approach is intended to deliver a product that meets several quality aspects: a system that conforms with the requirements, a system with high operational reliability, and source code that is easily readable.

Section II describes the Cleanroom approach and Section III presents a framework of goals for characterizing its effect. Section IV describes an empirical study using the approach. Section V gives the results of the analysis comparing projects developed using Cleanroom with those of a control group. The overall conclusions appear in Section VI.

II. Cleanroom Development

The following sections describe the Cleanroom software development approach, discuss its introduction to an environment, describe the relationship of Cleanroom to software prototyping, and explain the role of software tools in Cleanroom development.

A. Cleanroom Software Development

The IBM Federal Systems Division (FSD) [23], [19], [24], [21], [16] presents the Cleanroom software development method as a technical and organizational approach to developing software with certifiable reliability. The idea is to deny the entry of defects during the development of software, hence the term "Cleanroom." The focus of the method, which is an extension of the FSD software engineering program [22], is imposing discipline on the development process by integrating formal methods for specification and design, nonexecution-based program development, and statistically based independent testing. These components are intended to contribute to a software product that has a high probability of zero defects and consequently a high measure of operational reliability.

1) Software Life Cycle of Executable Increments: In the Cleanroom approach, software development is organized around the incremental development of the software product [16]. Instead of considering software design, implementation, and testing as sequential stages in a software life cycle, software development is considered as a sequence of executable product increments. The increments accumulate over the development life cycle and result in a final product with full functionality.

2) Formal Methods for Specification and Design: In order to support the life cycle of executable increments, Cleanroom developers utilize "structured specifications" to divide the product functionality into deeply nested sub-

Manuscript received February 28, 1985; revised May 30, 1986. This work was supported in part by the Air Force Office of Scientific Research under Contract AFOSR-F49620-80-C-001 to the University of Maryland and the University of California Faculty Research Fellowship Program. Computer support was provided in part by the Computer Science Center at the University of Maryland.

R. W. Selby is with the Department of Information and Computer Science, University of California, Irvine, CA 92717.

V. R. Basili and F. T. Baker are with the Department of Computer Science, University of Maryland, College Park, MD 20742.

IEEE Log Number 8716549.

0098-5589/87/0900-1027$01.00 © 1987 IEEE

sets that can be developed incrementally. The mathematically based design methodology in Cleanroom [22] incorporates the use of both structured specifications and state machine models [26]. A systems engineer introduces the structured specifications to restate the system requirements precisely and organize the complex problems into manageable parts [41]. The specifications determine the "system architecture" of the interconnections and groupings of capabilities to which state machine design practices can be applied. System implementation and test data formulation can then proceed from the structured specifications independently.

3) Development without Program Execution: The right-the-first-time programming methods used in Cleanroom are the ideas of functionally based programming in [38], [32]. The testing process is completely separated from the development process by not allowing the developers to test and debug their programs. The developers focus on the techniques of code reading by stepwise abstraction [32], code inspections [25], group walkthroughs [40], and formal verification [29], [32], [44], [20] to assert the correctness of their implementation. These nonexecution-based methods are referred to as "off-line software review techniques" in this paper. These constructive techniques apply throughout all phases of development, and condense the activities of defect detection and isolation into one operation. Empirical evaluations have suggested that the software review method of code reading by stepwise abstraction is at least as effective in detecting faults as execution-based methods [7], [43]. The intention in Cleanroom is to impose discipline on software development so that system correctness results from a coherent, readable design rather than from a reliance on execution-based testing. The notion that "Well, the software should always be tested to find the faults" is eliminated.

4) Statistically Based, Independent Testing: In the statistically based testing strategy of Cleanroom, independent testers simulate the operational environment of the system with random testing. This testing process includes defining the frequency distribution of inputs to the system, the frequency distribution of different system states, and the expanding range of developed system capabilities. Test cases then are chosen randomly and presented to the series of product increments, while concentrating on functions most recently delivered and maintaining the overall composite distribution of inputs. The independent testers then record observed failures and determine an objective measure of product reliability. Since software errors tend to vary widely in how frequently they are manifested as failures [1], operational testing is especially useful to assess the impact of software errors on product reliability. In addition to the statistical testing approach, the independent testers submit a limited number of test cases to ensure correct system operation for situations in which a software failure would be catastrophic. It is believed that the prior knowledge that a system will be evaluated by random testing will affect system reliability by enforcing a new discipline into the system developers.

The independent testing group operationally tests the software product increments from a perspective of reliability assessment, rather than a perspective of error detection. The responsibility of the test group is, therefore, to certify the reliability of the increments and final product rather than assist the development group in getting the product to an acceptable level of quality. One approach for measuring the reliability of the increments is through the use of a projected mean-time-between-failure (MTBF). MTBF estimations, based on user representative testing, provide both development managers and users with a useful, readily interpretable product reliability measure. Statistical models for calculating MTBF's projections include [34], [39], [33], [45], [15], [27], [16].

B. Introducing Cleanroom into a Development Environment

Before introducing the Cleanroom methodology into a software production environment, the developers need to be educated in the supporting technology areas. The technology areas consist of the development techniques and methods outlined in the above sections describing the components of Cleanroom. Potential Cleanroom users should also understand the goals of the development approach and be motivated to deliver high quality software products. One fundamental aspect of motivating the developers is to convince them that they can incorporate error prevention into the software process and actually produce error-free software. This "error-free perspective" is a departure from a current view that software errors are always present and error detection is the critical consideration.

C. Cleanroom versus Prototyping

The Cleanroom methodology and software prototyping are not mutually exclusive methods for developing software—the two approaches may be used together. The starting point for Cleanroom development is a document that states the user requirements. The production of that requirement document is an important portion of the software development process. Software prototyping is one approach that may be used to determine or refine the user requirements, and hence, produce the system requirements document [31], [47]. After the production of the requirements document, the prototype would be discarded and the Cleanroom methodology could be applied.

D. Tool Use in Cleanroom

Since Cleanroom developers do not execute their source code, does that mean that Cleanroom prohibits the use of tools during development? No—software tools can play an important role in the Cleanroom development approach. Various software tools can be used to help construct and manipulate the system design and source code. These tools can also be used to detect several types of errors that commonly occur in the system design and source code. The use of such tools facilitates the process of reviewing the system design and source code prior to

submission for testing by the independent group. Some of the tools that may assist Cleanroom developers include various static analyzers, data flow analyzers, syntax checkers, type checkers, formal verification checkers, concurrency analyzers, and modeling tools.

III. Investigation Goals

Some intriguing aspects of the Cleanroom approach include 1) development without testing and debugging of programs, 2) independent program testing for quality assurance (rather than to find faults or to prove "correctness" [30]), and 3) certification of system reliability before product delivery. In order to understand the effects of using Cleanroom, we proposed the following three goals: 1) characterize the effect of Cleanroom on the delivered product, 2) characterize the effect of Cleanroom on the software development process, and 3) characterize the effect of Cleanroom on the developers. An application of the goal/question/metric paradigm [6], [10] lead to the framework of goals and questions for this study which appears in Fig. 1. The empirical study executed to pursue these goals is described in the following section.

I. Characterize the effect of Cleanroom on the delivered product.
- A. For intermediate and novice programmers building a small system, what were the operational properties of the product?
  1. Did the product meet the system requirements?
  2. How did the operational testing results compare with those of a control group?
- B. What were the static properties of the product?
  1. Were the size properties of the product any different from what would be observed in a traditional development?
  2. Were the readability properties of the product any different?
  3. Was the control complexity any different?
  4. Was the data usage any different?
  5. Was the implementation language used differently?
- C. What contribution did programmer background have on the final product quality?

II. Characterize the effect of Cleanroom on the software development process.
- A. For intermediate and novice programmers building a small system, what techniques were used to prepare the developing system for testing submissions?
- B. What role did the computer play in development?
- C. Did the developers meet their delivery schedule?

III. Characterize the effect of Cleanroom on the developers.
- A. When intermediate and novice programmers built a small system, did the developers miss the satisfaction of executing their own programs?
  1. Did the missing of program execution have any relationship to programmer background or to aspects of the delivered product?
- B. How was the design and coding style of the developers affected by not being able to test and debug?
- C. Would the developers use Cleanroom again?

Fig. 1. Framework of goals and questions for Cleanroom development approach analysis.

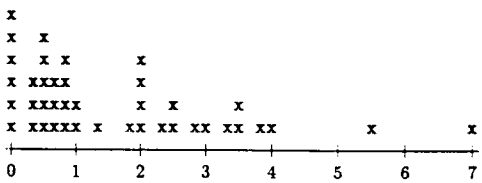


Fig. 2. Subjects' professional experience in years.

IV. Empirical Study Using Cleanroom

This section describes an empirical study comparing team projects developed using Cleanroom with those using a more conventional approach.

A. Subjects

Subjects for the empirical study came from the "Software Design and Development" course taught by F. T. Baker and V. R. Basili at the University of Maryland in the Falls of 1982 and 1983. The initial segment of the course was devoted to the presentation of several software development methodologies, including top-down design, modular specification and design, PDL, chief programmer teams, program correctness, code reading, walkthroughs, and functional and structural testing strategies. For the latter part of the course, the individuals were divided into three-person chief programmer teams for a group project [2], [37], [3]. We attempted to divide the teams equally according to professional experience, academic performance, and implementation language experience. The subjects had an average of 1.6 years professional experience and were university computer science students with graduate, senior, or junior standing. The subjects' professional experience predominantly came from government organizations and private software contractors in the Washington, DC area. Fig. 2 displays the distribution of the subjects' professional experience.

B. Project Developed

A requirements document for an electronic message system (read, send, mailing lists, authorized capabilities, etc.) was distributed to each of the teams. The project was to be completed in six weeks and was expected to be about 1500 lines of Simpl-T source code [9].[1] The development machine was a Univac 1100/82 running EXEC VIII, with 1200 baud interactive and remote access available.

C. Cleanroom Development Approach versus Traditional Approach

The ten teams in the Fall 1982 course applied the Cleanroom software development approach, while the five teams in the Fall 1983 course served as a control group (non-Cleanroom). All other aspects of the developments were the same. The two groups of teams were not statistically different in terms of professional experience, academic performance, or implementation language experience. If there were any bias between the two times the course was taught, it would be in favor of the 1983 (non-Cleanroom) group because the modular design portion of the course was presented earlier. It was also the second time F. T. Baker had taught the course.

The Cleanroom teams entered their source code on-line, used a syntax-checker (but did not do automated type-

[1]Simpl-T is a structured language that supports several string and file handling primitives, in addition to the usual control flow constructs available, for example, in Pascal. If Pascal or Fortran had been chosen, it would have been very likely that some individuals would have had extensive experience with the language, and this would have biased the comparison. Also, restricting access to a compiler that produced executable code would have been very difficult.

checking across modules), and were not able to execute their programs. The Cleanroom teams relied on the techniques of code reading, structured walkthroughs, and inspections to prepare their evolving systems before submission for independent testing. The non-Cleanroom teams were able to execute and debug their programs and applied several modern programming practices: modular design, top-down development, data abstraction, PDL, functional testing, design reviews, etc. The non-Cleanroom method was intended to reflect a software development approach that is currently in use in several software development organizations. Note that the non-Cleanroom method was roughly similar to the "disciplined team" development methodology examined in an earlier study [5].

One issue to consider when comparing a "newer" approach with an existing one is whether one group will try harder just because they are using the newer approach. This effect is referred to as the Hawthorne effect. In order to combat this potential effect, we decided to have all the members of one course apply the same development approach.[2] In order to diffuse any of the Cleanroom developers from thinking that they were being compared relative to a previously applied approach, we decided that Cleanroom would be used in the earlier (1982) course. Therefore, there was no obvious competing arrangement in terms of approaches that were newer versus controlled.

D. Project Milestones

The objective for all teams from both groups was to develop the full system described in the requirements document. The first document every team in either group turned in contained a system specification, composite design diagram, and implementation plan. The implementation plan was a series of milestones chosen by the individual teams which described when the various functions within the system would be available. At these various dates—minimum one week apart, maximum two—teams from the groups would then submit their systems for independent testing. Note that both the Cleanroom and non-Cleanroom teams had the benefit of the independent testing throughout development. An independent party would apply statistically based testing to each of the deliveries and report to the team members both the successful and unsuccessful test cases. The unsuccessful test cases would be included in a team's next test session for verification. The following section briefly describes the operationally based testing process applied to all projects by the independent tester.

E. Operational Testing of Projects

The testing approach used in Cleanroom is to simulate the developing system's environment by randomly selecting test data from an "operational profile," a frequency distribution of inputs to the system [46], [18]. The projects from both groups were tested interactively by an independent party (i.e., R. W. Selby) at the milestones chosen by each team. A distribution of inputs to the system was obtained by identifying the logical functions in the system and assigning each a frequency. This frequency assignment was accomplished by polling eleven well-seasoned users of a University of Maryland Vax 11/780 mailing system. Then test data were generated randomly from this profile and presented to the system. Recording of failure severity and times between failure took place during the testing process. The operational statistics referred to later were calculated from 50 user-session test cases run on the final system release of each team. For a complete explanation of the operationally based testing process applied to the projects, including test data selection, testing procedure, and failure observation, see [42].

F. Project Evaluation

All team projects were evaluated on their use of the particular software development techniques, the independent testing results, and a final oral interview. Both groups of subjects were judged to be highly motivated during the development of their systems. One reason for their motivation was their being graded based on the evaluation of their team projects. Information on the team projects was also collected from a background questionnaire, a post-development attitude survey, static source code analysis, and operating system statistics.

V. Data Analysis and Interpretation

The analysis and interpretation of the data collected from the study appear in the following sections, organized by the goal areas outlined earlier. In order to address the various questions posed under each of the goals, some raw data usually will be presented and then interpreted. Fig. 3 presents the number of source lines, executable statements, and procedures and functions to give a rough view of the systems developed.

A. Characterization of the Effect on the Product Developed

This section characterizes the differences between the products delivered by the two development groups. Researchers have delineated numerous perspectives of software product quality [36], [14], [13], and the following sections examine aspects of several of these perspectives. Initially we examine some operational properties of the products, followed by a comparison of some of their static properties.

1) Operational System Properties: In order to contrast the operational properties of the systems delivered by the two groups, both completeness of implementation and operational testing results were examined. A measure of implementation completeness was calculated by partitioning the required system into 16 logical functions (e.g., send mail to an individual, read a piece of mail, respond, add yourself to a mailing list, . . .). Each function in an implementation was then assigned a value of two if it completely met its requirements, a value of one if it par-

[2]This decision also happened to result in the two groups not being as close in terms of size as they could have been.

Team	Cleanroom	Source Lines	Executable Statments	Procedures & Functions
A	yes	1681	813	55
B	yes	1626	717	42
C	yes	1118	573	42
D	yes	1046	477	30
E	yes	1087	624	32
F	yes	1213	440	35
G	yes	1196	581	31
H	yes	1876	550	51
I	yes	1305	608	23
J	yes	1052	658	24
a	no	824	410	26
b	no	1429	633	18
c	no	2264	999	46
d	no	1629	626	67
e	no	1310	459	43

Fig. 3. System statistics.

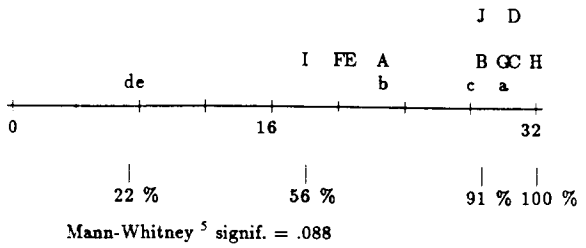


Fig. 4. Requirement conformance of the systems.

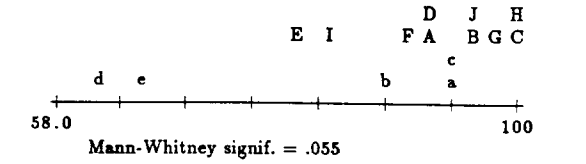


Fig. 5. Percentage of successful test cases during operational testing (without duplicate failures).

tially met them, or zero if it was inoperable. The total for each system was calculated; a maximum score of 32 was possible. Fig. 4 displays this subjective measure of requirement conformance for the systems. Note that in all figures presented, the ten teams using Cleanroom are in upper case and the five teams using a more conventional approach are in lower case. A first observation is that six of the ten Cleanroom teams built very close to the entire system. While not all of the Cleanroom teams performed equally well, a majority of them applied the approach effectively enough to develop nearly the whole product. More importantly, the Cleanroom teams met the requirements of the system more completely than did the non-Cleanroom teams.

To compare testing results among the systems developed in the two groups, 50 random user-session test cases were executed on the final release of each system to simulate its operational environment. If the final release of a system performed to expectations on a test case, the outcome was called a "success;" if not, the outcome was a "failure." If the outcome was a "failure" but the same failure was observed on an earlier test case run on the final release, the outcome was termed a "duplicate failure." Fig. 5 shows the percentage of successful test cases when duplicate failures are not included. The figure displays that Cleanroom projects had a higher percentage of successful test cases at system delivery.[3] When duplicate failures are included, however, the better performance of the Cleanroom systems is not nearly as significant (MW = 0.134).[4] This is caused by the Cleanroom projects having a relatively higher proportion of duplicate failures, even though they did better overall. This demonstrates that while reviewing the code, the Cleanroom developers focused less than the other group on certain parts of the system. The more uniform review of the whole system makes the performance of the system less sensitive to its operational profile. Note that operational environments of systems are usually difficult to define *a priori* and are subject to change.

In both of the product quality measures of implementation completeness and operational testing results, there was quite a variation in performance.[6] A wide variation may have been expected with an unfamiliar development technique, but the developers using a more traditional approach had a wider range of performance than did those using Cleanroom in both of the measures—even with there being twice as many Cleanroom teams. All of the above differences are magnified by recalling that the non-Cleanroom teams did not develop their systems in one monolithic step, they (also) had the benefit of periodic operational testing by independent testers. Since both groups of teams had independent testing of all their deliveries, the early testing of deliveries must have revealed most faults overlooked by the Cleanroom developers.

These comparisons suggest that the non-Cleanroom developers focused on a "perspective of the tester," sometimes leaving out classes of functions and causing a less completely implemented product and more (especially unique) failures. Off-line software review techniques, however, are more general and their use contributed to more complete requirement conformance and fewer failures in the Cleanroom products. In addition to examining the operational properties of the product, various static properties were compared.

2) Static System Properties: The first question in this goal area concerns the size of the final systems. Fig. 3 showed the number of source lines, executable statements, and procedures and functions for the various systems. The projects from the two groups were not statis-

[3]Although not considered here, various software reliability models have been proposed to forecast system reliability based on failure data (see Section II-A-4).

[4]To be more succinct, MW will sometimes be used to abbreviate the significance level of the Mann-Whitney statistic.

[5]The significance levels for the Mann-Whitney statistics reported are the probability of Type I error in a one-tailed test.

[6]An alternate perspective includes only the more successful projects from each group in the comparison of operational product quality. When the best 60 percent from each approach are examined (i.e., removing teams "d," "e," "A," "E," "F," and "I"), the Mann-Whitney significance level for comparing implementation completeness becomes 0.045 and the significance level for comparing successful test cases (without duplicate failures) becomes 0.034. Thus, comparing the best teams from each approach increases the evidence in favor of Cleanroom in both of these product quality measures.

tically different (MW > 0.10) in any of these three size attributes. Another question in this goal area concerns the readability of the delivered source code. Although readability is not equivalent to maintainability, modifiability, or reusability, it is a central component of each of these software quality aspects. Two aspects of reading and altering source code are the number of comments present and the density of the "complexity." In an attempt to capture the complexity density, syntactic complexity [4] was calculated and normalized by the number of executable statements. In addition to control-flow complexity, the syntactic complexity metric considers nesting depth and prime program decomposition [32]. The developers using Cleanroom wrote code that was more highly commented (MW = 0.089) and had a lower complexity density (MW = 0.079) than did those using the traditional approach. A calculation of either software science effort [28], cyclomatic complexity [35], or syntactic complexity without any size normalization, however, produced no significant differences (MW > 0.10). This seems as expected because all the systems were built to meet the same requirements.

Comparing the data usage in the systems, Cleanroom developers used a greater number of nonlocal data items (MW = 0.071). Also, Cleanroom projects possessed a higher percentage of assignment statements (MW = 0.056). These last two observations could be a manifestation of teaching the Cleanroom subjects modular design later in the course (see Section IV-C), or possibly an indication of using the approach. One interpretation of the Cleanroom developers' use of more nonlocal data could be that the resulting software would be less reusable and less portable. In fact, however, the increased use of nonlocal data by some Cleanroom developers was because of their use of data abstraction. In order to incorporate data abstraction into a system implemented in the Simpl-T programming language, developers may create independently compilable program units that have retained, nonlocal data and associated accessing routines.

Some interesting observations surface when the operational quality measures of just the Cleanroom products are correlated with the usage of the implementation language. Both percentage of successful test cases (without duplicate failures) and implementation completeness correlated with percentage of procedure calls (Spearman $R = 0.65$, signif. = 0.044, and $R = 0.57$, signif. = 0.08, respectively) and with percentage of IF statements ($R = 0.62$, signif. = 0.058, and $R = 0.55$, signif. = 0.10, respectively). However, both of these two product quality measures correlated negatively with percentage of CASE statements ($R = -0.86$, signif. = 0.001, and $R = -0.69$, signif. = 0.027, respectively) and with percentage of WHILE statements ($R = -0.65$, signif. = 0.044, and $R = -0.49$, signif. = 0.15, respectively). There were also some negative correlations between the product quality measures and the average software science effort per subroutine ($R = -0.52$, signif. = 0.12, and $R = -0.74$, signif. = 0.013, respectively) and the average number of occurrences of a variable ($R = -0.54$, signif. = 0.11, and $R = -0.56$, signif. = 0.09, respectively). Considering the products from all teams, both percentage of successful test cases (without duplicate failures) and implementation completeness had some correlation with percentage of IF statements ($R = 0.48$, signif. = 0.07, and $R = 0.45$, signif. = 0.09, respectively) and some negative correlation with percentage of CASE statements ($R = -0.48$, signif. = 0.07, and $R = -0.42$, signif. = 0.12, respectively). Neither of the operational product quality measures correlated with percentage of assignment statements when either all products or just Cleanroom products were considered. These observations suggest that the more successful Cleanroom developers simplified their use of the implementation language; i.e., they used more procedure calls and IF statements, used fewer CASE and WHILE statements, had a lower frequency of variable reuse, and wrote subroutines requiring less software science effort to comprehend.

3) Contribution of Programmer Background: When examining the contribution of the Cleanroom programmers' background to the quality of their final products, general programming language experience correlated with percentage of successful operational tests (without duplicate failures: Spearman $R = 0.66$, signif. = 0.04; with duplicates: $R = 0.70$, signif. = 0.03) and with implementation completeness ($R = 0.55$; signif. = 0.10). No relationship appears between either operational testing results or implementation completeness and either professional[7] or testing experience. These background/quality relations seem consistent with other studies [17].

4) Summary of the Effect on the Product Developed: In summary, Cleanroom developers delivered a product that 1) met system requirements more completely, 2) had a higher percentage of successful test cases, 3) had more comments and less dense control-flow complexity, and 4) used more nonlocal data items and a higher percentage of assignment statements. The more successful Cleanroom developers 1) used more procedure calls and IF statements, 2) used fewer CASE and WHILE statements, 3) reused variables less frequently, 4) developed subroutines requiring less software science effort to comprehend, and 5) had more general programming language experience.

B. Characterization of the Effect on the Development Process

In a postdevelopment attitude survey, the developers were asked how effectively they felt they applied off-line software review techniques in testing their projects (see Fig. 6). This was an attempt to capture some of the information necessary to answer the first question under this goal (question II-A). In order to make comparisons at the team level, the responses from the members of a team are

[7]In fact, there are very slight negative correlations between years of professional experience and both percentage of successful tests (without duplicate failures: $R = -0.46$, signif. = 0.18) and implementation completeness ($R = -0.47$, signif. = 0.17).

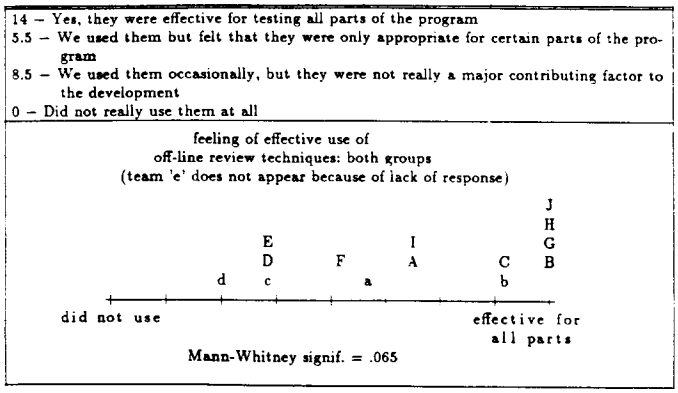


Fig. 6. Breakdown of responses to the attitude survey question, "Did you feel that you and your team members effectively used off-line review techniques in testing your project?" (Responses are from Cleanroom teams.)[8]

composed into an average for the team. The responses to the question appear on a team basis in a histogram in the second part of the figure. Of the Cleanroom developers, teams "A," "D," "E," "F," and "I" were the least confident in their use of the off-line review techniques and these teams also performed the worst in terms of operational testing results; four of these five teams performed the worst in terms of implementation completeness. Off-line review effectiveness correlated with percentage of successful operational tests (without duplicate failures) for the Cleanroom teams (Spearman $R = 0.74$; signif. = 0.014) and for all the teams ($R = 0.76$; signif. = 0.001); it correlated with implementation completeness for all the teams ($R = 0.58$; signif. = 0.023). Neither professional nor testing experience correlated with off-line review effectiveness when either all teams or just Cleanroom teams were considered.

The histogram in Fig. 6 shows that the Cleanroom developers felt they applied the off-line review techniques more effectively than did the non-Cleanroom teams. The non-Cleanroom developers were asked to give a relative breakdown of the amount of time spent applying testing and off-line review techniques. Their aggregate response was 39 percent off-line review, 52 percent functional testing, and 9 percent structural testing. From this breakdown, we observe that the non-Cleanroom teams primarily relied on functional testing to prepare their systems for independent testing. Since the Cleanroom teams were unable to rely on testing methods, they may have (felt they had) applied the off-line review techniques more effectively.

Since the role of the computer is more controlled when using Cleanroom, one would expect a difference in on-line activity between the two groups. Fig. 7 displays the amount of connect time that each of the teams cumulatively used. A comparison of the cpu-time used by the teams was less statistically significant (MW = 0.110). Neither of these measures of on-line activity related to

[8]There are half-responses because an individual checked both the second and third choices. The responses total to 28, not 30, because two separate teams lost a member late in the project. (See Section V-D).

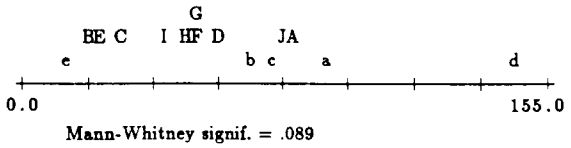


Fig. 7. Connect time in hours during project development.[9]

how effectively a team felt they had used the off-line review techniques when either all teams or just Cleanroom teams were considered. Although non-Cleanroom team "d" did a lot of on-line testing and non-Cleanroom team "e" did little, both teams performed poorly in the measures of operational product quality discussed earlier. The operating system of the development machine captured these system usage statistics. Note that the time the independent party spent testing is included.[10] These observations exhibit that Cleanroom developers spent less time on-line and used fewer computer resources. These results empirically support the reduced role of the computer in Cleanroom development.

Schedule slippage continues to be a problem in software development. It would be interesting to see whether the Cleanroom teams demonstrated any more discipline by maintaining their original schedules. All of the teams from both groups planned four releases of their evolving system, except for team "G" which planned five. Recall that at each delivery an independent party would operationally test the functions currently available in the system, according to the team's implementation plan. In Fig. 8, we observe that all the teams using Cleanroom kept to their original schedules by making all planned deliveries; only two non-Cleanroom teams made all their scheduled deliveries.

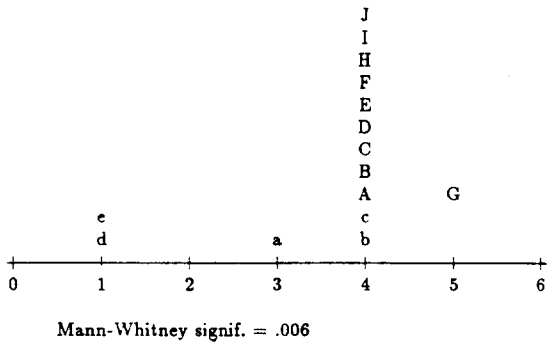
1) Summary of the Effect on the Development Process: Summarizing the effect on the development process, Cleanroom developers 1) felt they applied off-line review techniques more effectively, while non-Cleanroom teams focused on functional testing; 2) spent less time on-line and used fewer computer resources; and 3) made all their scheduled deliveries.

C. Characterization of the Effect on the Developers

The first question posed in this goal area is whether the individuals using Cleanroom missed the satisfaction of executing their own programs. Fig. 9 presents the responses to a question included in the postdevelopment attitude survey on this issue. As might be expected, almost all the individuals missed some aspect of program execution. As might not be expected, however, this missing of program execution had no relation to either the product quality measures mentioned earlier or the teams' professional or testing experience. Also, missing program execution did not increase with respect to program size (see Fig. 10).

[9]Non-Cleanroom team "e" entered a substantial portion of its system on a remote machine, only using the Univac computer mainly for compilation and execution. Team "e" was the only team that used any machine other than the Univac. (See Section V-D.)

[10]When the time the independent tester spent is not included, the significance levels for the nonparametric statistics do not change.

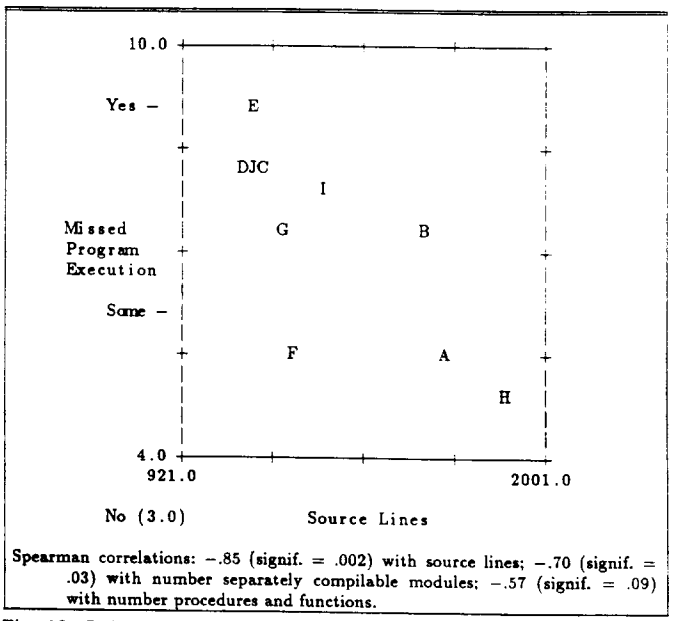


Mann-Whitney signif. = .006

Fig. 8. Number of system releases.

13 – Yes, I missed the satisfaction of program execution.
11 – I somewhat missed the satisfaction of program execution.
4 – No, I did not miss the satisfaction of program execution.

Fig. 9. Breakdown of responses to the attitude survey question, "Did you miss the satisfaction of executing your own programs?"



Spearman correlations: –.85 (signif. = .002) with source lines; –.70 (signif. = .03) with number separately compilable modules; –.57 (signif. = .09) with number procedures and functions.

Fig. 10. Relationship of program size versus missing program execution.

2 – Yes, my style was substantially revised.
15 – I modified some of my tendencies.
11 – It did not affect my style at all.
Frequently mentioned responses include
– kept design simple, attempted nothing fancy
– kept readability of code in mind
– already was a user of off-line review techniques
– very careful scrutiny of code for potential mistakes
– prepared for a larger range of inputs

Fig. 11. Breakdown of responses to the attitude survey question, "How was your design and coding style affected by not being able to test and debug?"

Fig. 11 displays the replies of the developers when they were asked how their design and coding style was affected by not being able to test and debug. At first it would seem surprising that more people did not modify their development style when applying the techniques of Cleanroom. Several persons mentioned, however, that they already utilized some of the ideas in Cleanroom. Keeping a simple design supports readability of the product and facilitates the processes of modification and verification. Although some of the objective product measures presented earlier showed differences in development style, these subjective ones are interesting and lend insight into actual programmer behavior.

One indicator of the impression that something new leaves on people is whether they would do it again. Fig. 12 presents the responses of the individuals when they were asked whether they would choose to use Cleanroom again as either a software development manager or as a programmer. Even though these responses were gathered (immediately) after course completion, subjects desiring to "please the instructor" may have responded favorably to this type of question regardless of their true feelings. Practically everyone indicated a willingness to apply the approach again. It is interesting to note that a greater number of persons in a managerial role would choose to always use it. Of the persons that ranked the reuse of Cleanroom fairly low in each category, four of the five were the same people. Of the six people that ranked reuse low, four were from less successful projects (one from team "A," one from team "E" and two from team "I"), but the other two came from reasonably successful developments (one from team "C" and one from team "J"). The particular individuals on teams "E," "I," and "J" were the four that rated reuse fairly low in both categories.

1) Summary of the Effect on the Developers: In summary of the effect on the developers, most Cleanroom developers 1) partially modified their development style, 2) missed program execution, and 3) indicated that they would use the approach again.

D. Distinction Among Teams

In spite of efforts to balance the teams according to various factors (see Section IV-A), a few differences among the teams were apparent. Two separate Cleanroom teams, "H" and "I," each lost a member late in the project. Thus at project completion, there were eight three-person and two two-person Cleanroom teams. Recall that team "H" performed quite well according to requirement conformance and testing results, while team "I" did poorly. Also, the second group of subjects did not divide evenly into three-person teams. Since one of those individuals had extensive professional experience, non-Cleanroom team "e" consisted of that one highly experienced person. Thus at project completion, there were four three-person and one one-person non-Cleanroom teams. Although team "e" wrote over 1300 source lines, this highly experienced person did not do as well as the other teams in some respects. This is consistent with another study in which teams applying a "disciplined methodology" in development outperformed individuals [5]. Appendix A contains the significance levels for the results of the analysis presented when team "e," when teams "H" and "I," and when teams "e," "H," and "I" are removed from the analysis. Removing teams "H" and "I" has

As a software development manager?
8 – Yes, at all times
14 – Yes, but only for certain projects
5 – Not at all
As a programmer?
4 – Yes, for all projects
18 – Yes, but not all the time
5 – Only if I had to
0 – I would leave if I had to

Fig. 12. Breakdown of responses to the attitude survey question, "Would you use Cleanroom again?" (One person did not respond to this question.)

little effect on the significance levels, while the removal of team "e" causes a decrease in all of the significance levels except for executable statements, software science effort, cyclomatic complexity, syntactic complexity, connect-time, and cpu-time.

VI. Conclusions

This paper describes "Cleanroom" software development—an approach intended to produce highly reliable software by integrating formal methods for specification and design, nonexecution-based program development, and statistically based independent testing. The goal structure, experimental approach, data analysis, and conclusions are presented for a replicated-project study examining the Cleanroom approach. This is the first investigation known to the authors that applied Cleanroom and characterized its effect relative to a more traditional development approach.

The data analysis presented and the testimony provided by the developers suggest that the major results of this study are the following. 1) Most of the developers were able to apply the techniques of Cleanroom effectively (six of the ten Cleanroom teams delivered at least 91 percent of the required system functions). 2) The Cleanroom teams' products met system requirements more completely and had a higher percentage of successful operationally generated test cases. 3) The source code developed using Cleanroom had more comments and less dense control-flow complexity. 4) The more successful Cleanroom developers modified their use of the implementation language; they used more procedure calls and IF statements, used fewer CASE and WHILE statements, and had a lower frequency of variable reuse (average number of occurrences per variable). 5) All ten Cleanroom teams made all of their scheduled intermediate product deliveries, while only two of the five non-Cleanroom teams did. 6) Although 86 percent of the Cleanroom developers indicated that they missed the satisfaction of program execution to some extent, this had no relation to the product quality measures of implementation completeness and successful operational tests. 7) Eighty-one percent of the Cleanroom developers said that they would use the approach again.

Based on the experience of applying Cleanroom in this study, some potential areas for improving the methodology are as follows. 1) As mentioned above, several Cleanroom developers tended to miss the satisfaction of program execution. In order to circumvent a potential long-term psychological effect, a method for providing such satisfaction to the developers would be useful. One suggestion would be for developers to witness, but not influence, program execution by the independent testers. 2) Several of the persons applying the Cleanroom approach mentioned that they had some difficulty visualizing the user interface, and hence, felt that the systems suffered in terms of "user-friendliness." One suggestion would be to prototype the user interfaces as part of the requirement determination phase, and then describe the interfaces in the requirements document, possibly using an interactive display specification language [11]. 3) A few of the Cleanroom developers said that they did not feel subjected to a "full test." Recall that the reliability certification component of the Cleanroom approach stands on the premise that operationally-based testing is sufficient to assess system reliability. One suggestion may be to augment the testing process with methods that enforce increased coverage of the system requirements, design, and implementation and/or methods that utilize frequent error profiles.

Overall, it seems that the ideas in Cleanroom help attain the goals of producing high quality software and increasing the discipline in the software development process. The complete separation of development from testing appears to cause a modification in the developers' behavior, resulting in increased process control and in more effective use of methods for software specification, design, off-line review, and verification. It seems that system modification and maintenance would be more easily done on a product developed in the Cleanroom method, because of the product's thoroughly conceived design and higher readability. Facilitating the software modification and maintenance tasks results in a corresponding reduction in associated costs to users. The amount of development effort required by the Cleanroom approach was not gathered in this study because its purpose was to examine the feasibility of Cleanroom and to characterize its effect. However, even if using Cleanroom required additional development effort, it seems that the potential reduction in maintenance and enhancement costs may result in an overall decrease in software life cycle cost. Thus, achieving high requirement conformance and high operational reliability coupled with low maintenance costs would help reduce overall costs, satisfy the user community, and support a long product lifetime.

Other studies which have compared software development methodologies include [5] and [12].[11] In [5] three software development approaches were compared: a disciplined-methodology team approach, an ad hoc team approach, and an ad hoc individual approach. The development approaches were applied by advanced university students comprising seven three-person teams, six three-person teams, and six individuals, respectively. They separately built a small (600–2200 line) compiler. The dis-

[11] For a survey of controlled, empirical studies that have been conducted in software engineering, see [8].

Measure	Average		Mann-Whitney significance levels			
	Clean-room Teams	Non-Clean-room Teams	All Teams	Without Team e	Without Teams H,I	Without Teams e,H,I
Source lines	1320.0	1491.2	.196	.240	.153	.198
Executable stmts	604.1	625.4	.500	.286	.442	.367
Procedures & functions	36.5	40.0	.357	.500	.330	.500
%Implementation completeness	82.5	60.0	.088	.197	.093	.196
%Successful tests (w/o duplicate failures)	92.5	80.8	.055	.128	.053	.116
%Successful tests (w/ duplicate failures)	78.7	59.2	.134	.285	.151	.304
Comments	194.9	122.2	.089	.102	.190	.198
Syntactic complexity/ executable stmts	1.5	1.6	.079	.179	.082	.175
Software Science E	6728.6e3	7355.4e3	.451	.240	.442	.248
Cyclomatic complexity	196.8	212.2	.250	.198	.255	.248
Syntactic complexity	917.5	1017.0	.500	.286	.500	.305
Non-local data items	37.6	24.2	.071	.129	.053	.117
%Assignment stmts	34.2	26.6	.056	.129	.040	.087
Off-line effectiveness	3.2	2.5	.065	.065	.098	.098
Connect-time (hr.)	41.0	71.3	.089	.012	.121	.021
Cpu-time (min.)	71.7	136.1	.110	.017	.072	.009
Deliveries	4.1	2.6	.006	.015	.010	.022

Fig. 13. Summary of measure averages and significance levels.

ciplined-methodology team approach significantly reduced the development costs as reflected in program changes and runs. The resulting designs from the disciplined-methodology teams and the ad hoc individuals were more coherent than the disjointed designs developed by the ad hoc teams. In [12] two software development approaches were compared: prototyping and specifying. Seven two- and three-person teams, consisting of university graduate students, developed separate versions of the same (2000–4000 line) application program. The systems developed by prototyping were smaller, required less development effort, and were easier to use. The systems developed by specifying had more coherent designs, more complete functionality, and software that was easier to integrate.

Future possible research directions include 1) assessment of the applicability of Cleanroom to larger software developments (note that aspects of the Cleanroom approach are being used in a 30 000 source line project [21], [16]); 2) empirical evaluation of the effect of Cleanroom from additional software quality perspectives, including reusability and modifiability; and 3) further characterization of the number and types of errors that occur when Cleanroom is or is not used.

This empirical study is intended to advance the understanding of the relationship between introducing discipline into the development process, as in Cleanroom, and several aspects of product quality: conformance with requirements, high operational reliability, and easily readable source code. The results given were calculated from a set of teams applying Cleanroom development on a relatively small project—the direct extrapolation of the findings to other projects and development environments is not implied.

Appendix A

Fig. 13 presents the measure averages and the significance levels for the above comparisons when team "e," when teams "H" and "I," and when teams "e," "H," and "I" are removed. The significance levels for the Mann-Whitney statistics reported are the probability of Type I error in a one-tailed test.

Acknowledgment

The authors are grateful to D. H. Hutchens and R. W. Reiter for the use of their static analysis program in this study.

References

[1] E. N. Adams, "Optimizing preventive service of software products," *IBM J. Res. Develop.*, vol. 28, no. 1, pp. 2–14, Jan. 1984.
[2] F. T. Baker, "Chief programmer team management of production programming," *IBM Syst. J.*, vol. 11, no. 1, pp. 131–149, 1972.
[3] —, "Chief programmer teams," in *Tutorial on Structured Programming: Integrated Practices*, V. R. Basili and F. T. Baker, Eds. New York: IEEE, 1981.
[4] V. R. Basili and D. H. Hutchens, "An empirical study of a syntactic metric family," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 664–672, Nov. 1983.
[5] V. R. Basili and R. W. Reiter, "A controlled experiment quantitatively comparing software development approaches," *IEEE Trans. Software Eng.*, vol. SE-7, May 1981.
[6] V. R. Basili and R. W. Selby, "Data collection and analysis in software research and management," in *Proc. Amer. Statist. Ass. and Biometric Soc. Joint Statist. Meetings*, Philadelphia, PA, August 13–16, 1984.
[7] —, "Comparing the effectiveness of software testing strategies," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1501, May 1985; to appear in *IEEE Trans. Software Eng.*
[8] V. R. Basili, R. W. Selby, and D. H. Hutchens, "Experimentation in software engineering," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 733–743, July 1986.
[9] V. R. Basili and A. J. Turner, *SIMPL-T: A Structured Programming Language*. Geneva, IL: Paladin House, 1976.

[10] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 728–738, Nov. 1984.
[11] L. J. Bass, "An approach to user specification of interactive display interfaces," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 686–698, Aug. 1985.
[12] B. W. Boehm, T. E. Gray, and T. Seewaldt, "Prototyping versus specifying: A multiproject experiment," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 290–303, May 1984.
[13] T. P. Bowen, G. B. Wigle, and J. T. Tsai, "Specification of software quality attributes," Rome Air Development Center, Griffiss Air Force Base, NY, Tech. Rep. RADC-TR-85-37 (3 volumes), Feb. 1985.
[14] J. P. Cavano and J. A. McCall, "A framework for the measurement of software quality," in *Proc. Software Quality and Assurance Workshop*, San Diego, CA, Nov. 1978, pp. 133–139.
[15] P. A. Currit, "Cleanroom certification model," in *Proc. 8th Annu. Software Eng. Workshop*, NASA/GSFC, Greenbelt, MD, Nov. 1983.
[16] P. A. Currit, M. Dyer, and H. D. Mills, "Certifying the reliability of software," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 3–11, Jan. 1986.
[17] B. Curtis, "Cognitive science of programming," in *Sixth Minnowbrook Workshop Software Performance Evaluation*, Blue Mountain Lake, NY, July 19–22, 1983.
[18] J. W. Duran and S. Ntafos, "A report on random testing*," in *Proc. Fifth Int. Conf. Software Eng.*, San Diego, CA, Mar. 9–12, 1981, pp. 179–183.
[19] M. Dyer, "Cleanroom software development method," IBM Federal Systems Division, Bethesda, MD, Oct. 14, 1982.
[20] ——, "Software validation in the Cleanroom development method," IBM-FSD Tech. Rep. 86.0003, Aug. 19, 1983.
[21] ——, "Software development under statistical quality control," in *Proc. NATO Advanced Study Institute: The Challenge of Advanced Computing Technology to System Design Methods*, Durham, UK, July 29–Aug. 10, 1985.
[22] M. Dyer, R. C. Linger, H. D. Mills, D. O'Neill, and R. E. Quinnan, "The management of software engineering," *IBM Syst. J.*, vol. 19, no. 4, 1980.
[23] M. Dyer and H. D. Mills, "The Cleanroom approach to reliable software development," in *Proc. Validation Methods Research for Fault-Tolerant Avionics and Control Systems Sub-Working-Group Meeting: Production of Reliable Flight-Crucial Software*, Research Triangle Institute, NC, Nov. 2–4, 1981.
[24] ——, "Developing electronic systems with certifiable reliability," in *Proc. NATO Conf.*, Summer 1982.
[25] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 15, no. 3, pp. 182–211, 1976.
[26] A. B. Ferrentino and H. D. Mills, "State machines and their semantics in software engineering," in *Proc. IEEE COMPSAC*, 1977.
[27] A. L. Goel, "A guidebook for software reliability assessment," Dep. Industrial Eng. and Oper. Res., Syracuse Univ., New York, Tech. Rep. 83-11, Apr. 1983.
[28] M. H. Halstead, *Elements of Software Science*. New York: North-Holland, 1977.
[29] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–583, Oct. 1969.
[30] W. E. Howden, "Reliability of the path analysis testing strategy," *IEEE Trans. Software Eng.*, vol. SE-2, no. 3, Sept. 1976.
[31] P. Kerola and P. Freeman, "A comparison of lifecycle models," in *Proc. 5th Int. Conf. Software Eng.*, Mar. 1981, pp. 90–99.
[32] R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*. Reading, MA: Addison-Wesley, 1979.
[33] B. Littlewood, "Stochastic reliability growth: A model for fault renovation computer programs and hardware designs," *IEEE Trans. Rel.*, vol. R-30, Oct. 1981.
[34] B. Littlewood and J. L. Verrall, "A Bayesian reliability growth model for computer software," *Appl. Statist.*, vol. 22, no. 3,1973.
[35] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 308–320, Dec. 1976.
[36] J. A. McCall, P. Richards, and G. Walters, "Factors in software quality," Rome Air Development Center, Griffiss Air Force Base, NY, Tech. Rep. RADC-TR-77-369, Nov. 1977.
[37] H. D. Mills, "Chief programmer teams: Principles and procedures," IBM Corp., Gaithersburg, MD, Rep. FSC 71-6012, 1972.
[38] ——, "Mathematical foundations for structural programming," IBM Rep. FSL 72-6021, 1972.
[39] J. D. Musa, "A theory of software reliability and its application," *IEEE Trans. Software Eng.*, vol. SE-1, no. 3, pp. 312–327, 1975.
[40] G. J. Myers, *Software Reliability: Principles & Practices*. New York: Wiley, 1976.
[41] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.
[42] R. W. Selby, "Evaluations of software technologies: Testing, CLEANROOM, and metrics," Ph.D. dissertation, Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1500, 1985.
[43] ——, "Combining software testing strategies: An empirical evaluation," in *Proc. Workshop Software Testing*, Banff, Alta., Canada, July 15–17, 1986, pp. 82–91.
[44] K. S. Shankar, "A functional approach to module verification," *IEEE Trans. Software Eng.*, vol. SE-8, Mar. 1982.
[45] J. G. Shanthikumar, "A statistical time dependent error occurrence rate software reliability model with imperfect debugging," in *Proc. 1981 Nat. Comput. Conf.*, June 1981.
[46] R. A. Thayer, M. Lipow, and E. C. Nelson, *Software Reliability*. Amsterdam, The Netherlands: North-Holland, 1978.
[47] M. V. Zelkowitz and M. Branstad, in *Proc. ACM SIGSOFT Rapid Prototyping Symp.*, Apr. 1982.

Richard W. Selby (S'83–M'85) received the B.A. degree in mathematics and computer science from Saint Olaf College, Northfield, MN, in 1981 and the M.S. and Ph.D. degrees in computer science from the University of Maryland, College Park, in 1983 and 1985, respectively.

He is an Assistant Professor of Information and Computer Science at the University of California, Irvine. His research interests include methodologies for developing and testing software, techniques for empirically evaluating software methodologies, and software environments.

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

Victor R. Basili (M'83–SM'84) is Professor and Chairman of the Computer Science Department at the University of Maryland, College Park. He was involved in the design and development of several software projects, including the SIMPL family of programming languages. He is currently measuring and evaluating software development in industrial settings and has consulted with many agencies and organizations, including IBM, GE, CSC, GTE, MCC, AT&T Bell Laboratories, NRL, NSWC, and NASA. He is one of the founders and principals in the Software Engineering Laboratory, a joint venture established in 1976 between NASA/Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation. In this context he has worked closely with CSE in developing models and metrics for the software development process and product. He has authored over 70 published papers on the methodology, the quantitative analysis, and the evaluation of the software development process and product. In 1982 he received the Outstanding Paper Award from the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING for his paper on the evaluation of methodologies.

Dr. Basili was Program Chairman for the 6th International Conference on Software Engineering, and the First ACM SIGSOFT Software Engineering Symposium on Tools and Methodology Evaluation. He serves on the editorial boards of the *Journal of Systems and Software* and the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. He is a member of the Association for Computing Machinery and the Executive Committee of the Technical Committee on Software Engineering, and is a senior member of the IEEE Computer Society.

F. Terry Baker, photograph and biography not available at the time of publication.