Experimentation in Software Engineering

VICTOR R. BASILI, SENIOR MEMBER, IEEE, RICHARD W. SELBY, MEMBER, IEEE, AND DAVID H. HUTCHENS, MEMBER, IEEE

Abstract—**Experimentation in software engineering supports the advancement of the field through an iterative learning process. In this paper we present a framework for analyzing most of the experimental work performed in software engineering over the past several years. We describe a variety of experiments in the framework and discuss their contribution to the software engineering discipline. Some useful recommendations for the application of the experimental process in software engineering are included.**

Index Terms—**Controlled experiment, data collection and analysis, empirical study, experimental design, software metrics, software technology measurement and evaluation.**

I. Introduction

As any area matures, there is the need to understand its components and their relationships. An experimental process provides a basis for the needed advancement in knowledge and understanding. Since software engineering is in its adolescence, it is certainly a candidate for the experimental method of analysis. Experimentation is performed in order to help us better evaluate, predict, understand, control, and improve the software development process and product.

Experimentation in software engineering, as with any other experimental procedure, involves an iteration of a hypothesize and test process. Models of the software process or product are built, hypotheses about these models are tested, and the information learned is used to refine the old hypotheses or develop new ones. In an area like software engineering, this approach takes on special importance because we greatly need to improve our knowledge of how software is developed, the effect of various technologies, and what areas most need improvement. There is a great deal to be learned and intuition is not always the best teacher.

In this paper we lay out a framework for analyzing most of the experimental work that has been performed in software engineering over the past several years. We then discuss a variety of these experiments, their results, and the impact they have had on our knowledge of the software engineering discipline.

Manuscript received July 15, 1985; revised January 15, 1986. This work was supported in part by the Air Force Office of Scientific Research under Contract AFOSR-F49620-80-C-001 and by the National Aeronautics and Space Administration under Grant NSG-5123 to the University of Maryland. Computer support was provided in part by the Computer Science Center at the University of Maryland.

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

R. W. Selby was with the Department of Computer Science, University of Maryland, College Park, MD 20742. He is now with the Department of Information and Computer Science, University of California, Irvine, CA 92717.

D. H. Hutchens is with the Department of Computer Science, Clemson University, Clemson, SC 29634.

IEEE Log Number 8608188.

II. Objectives

There are three overall goals for this work. The first objective is to describe a framework for experimentation in software engineering. The framework for experimentation is intended to help structure the experimental process and to provide a classification scheme for understanding and evaluating experimental studies. The second objective is to classify and discuss a variety of experiments from the literature according to the framework. The description of several software engineering studies is intended to provide an overview of the knowledge resulting from experimental work, a summary of current research directions, and a basis for learning from past experience with experimentation. The third objective is to identify problem areas and lessons learned in experimentation in software engineering. The presentation of problem areas and lessons learned is intended to focus attention on general trends in the field and to provide the experimenter with useful recommendations for performing future studies. The following three sections address these goals.

III. Experimentation Framework

The framework of experimentation, summarized in Fig. 1, consists of four categories corresponding to phases of the experimentation process: 1) definition, 2) planning, 3) operation, and 4) interpretation. The following sections discuss each of these four phases.

A. Experiment Definition

The first phase of the experimental process is the study definition phase. The study definition phase contains six parts: 1) motivation, 2) object, 3) purpose, 4) perspective, 5) domain, and 6) scope. Most study definitions contain each of the six parts; an example definition appears in Fig. 2.

There can be several motivations, objects, purposes, or perspectives in an experimental study. For example, the motivation of a study may be to understand, assess, or improve the effect of a certain technology. The "object of study" is the primary entity examined in a study. A study may examine the final software product, a development process (e.g., inspection process, change process), a model (e.g., software reliability model), etc. The

0098-5589/86/0700-0733$01.00 © 1986 IEEE

I. Definition					
Motivation	Object	Purpose	Perspective	Domain	Scope
Understand Assess Manage Engineer Learn Improve Validate Assure	Product Process Model Metric Theory	Characterize Evaluate Predict Motivate	Developer Modifier Maintainer Project manager Corporate manager Customer User Researcher	Programmer Program/project	Single project Multi-project Replicated project Blocked subject-project

II. Planning		
Design	Criteria	Measurement
Experimental designs Incomplete block Completely randomized Randomized block Fractional factorial Multivariate analysis Correlation Factor analysis Regression Statistical models Non-parametric Sampling	Direct reflections of cost/quality Cost Errors Changes Reliability Correctness Indirect reflections of cost/quality Data coupling Information visibility Programmer comprehension Execution coverage Size Complexity	Metric definition Goal-question-metric Factor-criteria-metric Metric validation Data collection Automatability Form design and test Objective vs. subjective Level of measurement Nominal/classificatory Ordinal/ranking Interval Ratio

III. Operation		
Preparation	Execution	Analysis
Pilot study	Data collection Data validation	Quantitative vs. qualitative Preliminary data analysis Plots and histograms Model assumptions Primary data analysis Model application

IV. Interpretation		
Interpretation context	Extrapolation	Impact
Statistical framework Study purpose Field of research	Sample representativeness	Visibility Replication Application

Fig. 1. Summary of the framework of experimentation.

Definition element	example
Motivation	To improve the unit testing process,
Purpose	characterize and evaluate
Object	the processes of functional and structural testing
Perspective	from the perspective of the developer
Domain: programmer	as they are applied by experienced programmers
Domain: program	to unit-size software
Scope	in a blocked subject-project study.

Fig. 2. Study definition example.

#Teams per project	#Projects: one	#Projects: more than one
one	Single project	Multi-project variation
more than one	Replicated project	Blocked subject-project

Fig. 3. Experimental scopes.

purpose of a study may be to characterize the change in a system over time, to evaluate the effectiveness of testing processes, to predict system development cost by using a cost model, to motivate[1] the validity of a theory by analyzing empirical evidence, etc. In experimental studies that examine "software quality," the interpretation usually includes correctness if it is from the perspective of a developer or reliability if it is from the perspective of a customer. Studies that examine metrics for a given project type from the perspective of the project manager may interest certain project managers, while corporate managers may only be interested if the metrics apply across several project types.

[1] For clarification, the usage of the word "motivate" as a study purpose is distinct from the study "motivation."

Two important domains that are considered in experimental studies of software are 1) the individual programmers or programming teams (the "teams") and 2) the programs or projects (the "projects"). "Teams" are (possibly single-person) groups that work separately, and "projects" are separate programs or problems on which teams work. Teams may be characterized by experience, size, organization, etc., and projects may be characterized by size, complexity, application, etc. A general classification of the scopes of experimental studies can be obtained by examining the sizes of these two domains considered (see Fig. 3). Blocked subject–project studies examine one or more objects across a set of teams and a set of projects. Replicated project studies examine ob-

ject(s) across a set of teams and a single project, while multiproject variation studies examine object(s) across a single team and a set of projects. Single project studies examine object(s) on a single team and a single project. As the representativeness of the samples examined and the scope of examination increase, the wider-reaching a study's conclusions become.

B. Experiment Planning

The second phase of the experimental process is the study planning phase. The following sections discuss aspects of the experiment planning phase: 1) design, 2) criteria, and 3) measurement.

The design of an experiment couples the study scope with analytical methods and indicates the domain samples to be examined. Fractional factorial or randomized block designs usually apply in blocked subject–project studies, while completely randomized or incomplete block designs usually apply in multiproject and replicated project studies [33], [41]. Multivariate analysis methods, including correlation, factor analysis, and regression [75], [80], [89], generally may be used across all experimental scopes. Statistical models may be formulated and customized as appropriate [89]. Nonparametric methods should be planned when only limited data may be available or distributional assumptions may not be met [100]. Sampling techniques [40] may be used to select representative programmers and programs/projects to examine.

Different motivations, objects, purposes, perspectives, domains, and scopes require the examination of different criteria. Criteria that tend to be direct reflections of cost/quality include cost [114], [108], [86], [5], [28], errors/changes [49], [24], [112], [2], [81], [13], reliability [42], [64], [56], [69], [70], [76], [77], [95], and correctness [51], [61], [68]. Criteria that tend to be indirect reflections of cost/quality include data coupling [62], [48], [104], [78], information visibility [85], [83], [55], programmer understanding [99], [103], [109], [113], execution coverage [105], [15], [18], and size/complexity [11], [59], [71].

The concrete manifestations of the cost/quality aspects examined in the experiment are captured through measurement. Paradigms assist in the metric definition process: the goal-question-metric paradigm [17], [25], [19], [93] and the factor-criteria-metric paradigm [39], [72]. Once appropriate metrics have been defined, they may be validated to show that they capture what is intended [7], [21], [45], [50], [108], [116]. The data collection process includes developing automated collection schemes [16] and designing and testing data collection forms [25], [27]. The required data may include both objective and subjective data and different levels of measurement: nominal (or classificatory), ordinal (or ranking), interval, or ratio [100].

C. Experiment Operation

The third phase of the experimental process is the study operation phase. The operation of the experiment consists of 1) preparation, 2) execution, and 3) analysis. Before conducting the actual experiment, preparation may include a pilot study to confirm the experimental scenario, help organize experimental factors (e.g., subject expertise), or inoculate the subjects [45], [44], [63], [18], [113], [73]. Experimenters collect and validate the defined data during the execution of the study [21], [112]. The analysis of the data may include a combination of quantitative and qualitative methods [30]. The preliminary screening of the data, probably using plots and histograms, usually precedes the formal data analysis. The process of analyzing the data requires the investigation of any underlying assumptions (e.g., distributional) before the application of the statistical models and tests.

D. Experiment Interpretation

The fourth phase of the experimental process is the study interpretation phase. The interpretation of the experiment consists of 1) interpretation context, 2) extrapolation, and 3) impact. The results of the data analysis from a study are interpreted in a broadening series of contexts. These contexts of interpretation are the statistical framework in which the result is derived, the purpose of the particular study, and the knowledge in the field of research [16]. The representativeness of the sampling analyzed in a study qualifies the extrapolation of the results to other environments [17]. Several follow-up activities contribute to the impact of a study: presenting/publishing the results for feedback, replicating the experiment [33], [41], and actually applying the results by modifying methods for software development, maintenance, management, and research.

IV. Classification of Analyses

Several investigators have published studies in the four general scopes of examination: blocked subject–project, replicated project, multiproject variation, or single project. The following sections cite studies from each of these categories. Note that surveys on experimentation methodology in empirical studies include [35], [96], [74], [98]. Each of the sections first discusses one experiment in moderate depth, using italicized keywords from the framework for experimentation, and then chronologically presents an overview of several others in the category. In any survey of this type it is almost certain that some deserving work has been accidentally omitted. For this, we apologize in advance.

A. Blocked Subject–Project Studies

With a *motivation* to improve and better understand unit testing, Basili and Selby [18] conducted a study whose *purpose* was to characterize and evaluate the processes (i.e., *objects*) of code reading, functional testing, and structural testing from the *perspective* of the developer. The testing processes were examined in a blocked subject–project *scope*, where 74 student through professional programmers (from the programmer *domain*) tested four unit-size programs (from the program *domain*) in a rep-

licated fractional factorial *design*. Objective *measurement* of the testing processes was in several *criteria* areas: fault detection effectiveness, fault detection cost, and classes of faults detected. Experiment *preparation* included a pilot study [63], *execution* incorporated both manual and automated monitoring of testing activity, and *analysis* used analysis of variance methods [33], [90]. The major results (in the *interpretation context* of the study purpose) included: 1) with the professionals, code reading detected more software faults and had a higher fault detection rate than did the other methods; 2) with the professionals, functional testing detected more faults than did structural testing, but they were not different in fault detection rate; 3) with the students, the three techniques were not different in performance, except that structural testing detected fewer faults than did the others in one study phase; and 4) overall, code reading detected more interface faults and functional testing detected more control faults than did the other methods. A major result (in the *interpretation context* of the field of research) was that the study suggested that nonexecution based fault detection, as in code reading, is at least as effective as on-line methods. The particular programmers and programs sampled qualify the *extrapolation* of the results. The *impact* of the study was an advancement in the understanding of effective software testing methods.

In order to understand program debugging, Gould and Drongowski [58] evaluated several related factors, including effect of debugging aids, effect of fault type, and effect of particular program debugged from the perspective of the developer and maintainer. Thirty experienced programmers independently debugged one of four one-page programs that contained a single fault from one of three classes. The major results of these studies were: 1) debugging is much faster if the programmer has had previous experience with the program, 2) assignment bugs were harder to find than other kinds, and 3) debugging aids did not seem to help programmers debug faster. Consistent results were obtained when the study was conducted on ten additional experienced programmers [57]. These results and the identification of possible "principles" of debugging contributed to the understanding of debugging methodology.

In order to improve experimentation methodology and its application, Weissman [113] evaluated programmers' ability to understand and modify a program from the perspective of the developer and modifier. Various measures of programmer understanding were calculated, in a series of factorial design experiments, on groups of 16–48 university students performing tasks on two small programs. The study emphasized the need for well-structured and well-documented programs and provided valuable testimony on and worked toward a suitable experimentation methodology.

In order to assess the impact of language features on the programming process, Gannon and Horning [54] characterized the relationship of language features to software reliability from the perspective of the developer. Based on an analysis of the deficiencies in a programming language, nine different features were modified to produce a new version. Fifty-one advanced students were divided into two groups and asked to complete implementations of two small but sophisticated programs (75–200 line) in the original language and its modified version. The redesigned features in the two languages were contrasted in program fault frequency, type, and persistence. The experiment identified several language-design decisions that significantly affected reliability, which contributed to the understanding of language design for reliable software.

In order to understand the unit testing process better, Hetzel [60] evaluated a reading technique and functional and "selective" testing (a composite approach) from the perspective of the developer. Thirty-nine university students applied the techniques to three unit-size programs in a Latin square design. Functional and "selective" testing were equally effective and both superior to the reading technique, which contributed to our understanding of testing methodology.

In order to improve and better understand the maintenance process, Curtis *et al.* [44] conducted two experiments to evaluate factors that influence two aspects of software maintenance, program understanding, and modification, from the perspective of the developer and maintainer. Thirty-six junior through advanced professional programmers in each experiment examined three classes of small (36–57 source line) programs in a factorial design. The factors examined include control flow complexity, variable name mnemonicity, type of modification, degree of commenting, and the relationship of programmer performance to various complexity metrics. In [45] they continued the investigation of how software characteristics relate to psychological complexity and presented a third experiment to evaluate the ability of 54 professional programmers to detect program bugs in three programs in a factorial design. The series of experiments suggested that software science [59] and cyclomatic complexity [71] measures were related to the difficulty experienced by programmers in locating errors in code.

In order to improve and better understand program debugging, Weiser [110] evaluated the theory that "programmers use 'slicing' (stripping away a program's statements that do not influence a given variable at a given statement) when debugging" from the perspective of the developer, maintainer, and researcher. Twenty-one university graduate students and programming staff debugged a fault in three unit-size (75–150 source line) programs in a nonparametric design. The study results supported the slicing theory, that is, programmers during debugging routinely partitioned programs into a coherent, discontiguous piece (or slice). The results advanced the understanding of software debugging methodology.

In order to improve design techniques, Ramsey, Atwood, and Van Doren [87] evaluated flowcharts and program design languages (PDL) from the perspective of the developer. Twenty-two graduate students designed two small (approximately 1000 source line) projects, one using

flowcharts and the other using PDL. Overall, the results suggested that design performance and designer–programmer communication were better for projects using PDL.

In order to validate a theory of programming knowledge, Soloway and Ehrlich [102] conducted two studies, using 139 novices and 41 professional programmers, to evaluate programmer behavior from the perspective of the researcher. The theory was that programming knowledge contained programming plans (generic program fragments representing common sequences of actions) and rules of programming discourse (conventions used in composing plans into programs). The results supported the existence and use of such plans and rules by both novice and advanced programmers.

Other blocked subject–project studies include [82], [115], and [111].

B. Replicated Project Studies

With a *motivation* to assess and better understand team software development methodologies, Basili and Reiter [16] conducted a study whose *purpose* was to characterize and evaluate the development processes (i.e., *objects*) of a 1) disciplined-methodology team approach, 2) ad hoc team approach, and 3) ad hoc individual approach from the *perspective* of the developer and project manager. The development processes were examined in a replicated project *scope*, in which advanced university students comprising seven three-person teams, six three-person teams, and six individuals (from the programmer *domain*) used the approaches, respectively. They separately developed a small (600–2200 line) compiler (from the program *domain*) in a nonparametric *design*. Objective *measurement* of the development approaches was in several *criteria* areas: number of changes, number of program runs, program data usage, program data coupling/binding, static program size/complexity metrics, language usage, and modularity. Experiment *preparation* included presentation of relevant material [68], [8], [34], *execution* included automated monitoring of on-line development activity and *analysis* used nonparametric comparison methods. The major results (in the *interpretation context* of the study purpose) included: 1) the methodological discipline was a key influence on the general efficiency of the software development process; 2) the disciplined team methodology significantly reduced the costs of software development as reflected in program runs and changes; and 3) the examination of the effect of the development approaches was accomplished by the use of quantitative, objective, unobtrusive, and automatable process and product metrics. A major result (in the *interpretation context* of the field of research) was that the study supported the belief that incorporating discipline in software development reflects positively on both the development process and final product. The particular programmers and program sampled qualify the *extrapolation* of the results. The *impact* of the study was an advancement in the understanding of software development methodologies and their evaluation.

In order to improve the design and implementation processes, Parnas [84] evaluated system modularity from the perspective of the developer. Twenty university undergraduates each developed one of four different types of implementations for one of five different small modules. Then each of the modules were combined with others to form several versions of the whole system. The results were that minor effort was required in assembling the systems and that major system changes were confined to small, well-defined subsystems. The results supported the ideas on formal specifications and modularity discussed in [83] and [85], and advanced the understanding of design methodology.

In order to assess the impact of static typing of programming languages in the development process, Gannon [53] evaluated the use of a statically typed language (having integers and strings) and a "typeless" language (e.g., arbitrary subscripting of memory) from the perspective of the developer. Thirty-eight students programmed a small (48–297 source line) problem in both languages, with half doing it in each order. The two languages were compared in the resulting program faults, the number of runs containing faults, and the relation of subject experience to fault proneness. The major result was that the use of a statically typed language can increase programming reliability, which improved our understanding of the design and use of programming languages.

In order to improve program composition, comprehension, debugging, and modification, Shneiderman [99] evaluated the use of detailed flowcharts in these tasks from the perspective of the developer, maintainer, modifier, and researcher. Groups of 53–70 novice through intermediate subjects, in a series of five experiments, performed various tasks using small programs. No significant differences were found between groups that used and those that did not use flowcharts, questioning the merit of using detailed flowcharts.

In order to improve and better understand the unit testing process, Myers [79] evaluated the techniques of three-person walk-throughs, functional testing, and a control group from the perspective of the developer. Fifty-nine junior through advanced professional programmers applied the techniques to test a small (100 source line) but nontrivial program. The techniques were not different in the number of faults they detected, all pairings of techniques were superior to single techniques, and code reviews were less cost-effective than the others. These results improved our understanding of the selection of appropriate software testing techniques.

In order to validate a particular metric family, Basili and Hutchens [11] evaluated the ability of a proposed metric family to explain differences in system development methodologies and system changes from the perspective of the developer, project manager, and researcher. The metrics were applied to 19 versions of a small (600–2200) compiler, which were developed by

teams of advanced university students using three different development approaches (see the first study [16] described in this section). The major results included: 1) the metrics were able to differentiate among projects developed with different development methodologies; and 2) the differences among individuals had a large effect on the relationships between the metrics and aspects of system development. These results provided insights into the formulation and appropriate use of software metrics.

In order to improve the understanding of why software errors occur, Soloway *et al.* [65], [101] characterized programmer misconceptions, cognitive strategies, and their manifestations as bugs in programs from the perspective of the developer and researcher. Two hundred and four novice programmers separately attempted implementations of an elementary program. The results supported the programmers' intended use of "programming plans" [103] and revealed that most people preferred a read–process strategy over a process–read strategy. The results advanced the understanding of how individuals write programs, why they sometimes make errors, and what programming language constructs should be available.

In order to understand the effect of coding conventions on program comprehensibility, Miara *et al.* [73] conducted a study to evaluate the relationship between indentation levels and program comprehension from the perspective of the developer. Eighty-six novice through professional subjects answered questions about one of seven program variations with different level and type of indentation. The major result was that an indentation level of two or four spaces was preferred over zero or six spaces.

In order to improve software development approaches, Boehm, Gray, and Seewaldt [29] characterized and evaluated the prototyping and specifying development approaches from the perspective of the developer, project manager, and user. Seven two- and three-person teams, consisting of university graduate students, developed versions of the same application software system (2000–4000 line); four teams used a requirement/design specifying approach and three teams used a prototyping approach. 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. These results contributed to the understanding of the merits and appropriateness of software development approaches.

In order to validate the theoretical model for *N*-version programming [3], [66], Knight and Leveson [67] conducted a study to evaluate the effectiveness of *N*-version programming for reliability from the perspective of the customer and user. *N*-version programming uses a high-level driver to connect several separately designed versions of the same system, the systems "vote" on the correct solution, and the solution provided by the majority of the systems is output. Twenty-seven graduate students were asked to independently design an 800 source line system. The factors examined included individual system reliability, total *N*-version system reliability, and classes of faults that occurred in systems simultaneously. The major result was that the assumption of independence of the faults in the programs was not justified, and therefore, the reliability of the combined "voting" system was not as high as given by the model.

In order to improve and better understand software development approaches, Selby, Basili, and Baker [94] characterized and evaluated the Cleanroom development approach [46], [47], in which software is developed without execution (i.e., completely off-line), from the perspective of the developer, project manager, and customer. Fifteen three-person teams of advanced university students separately developed a small system (800–2300 source line); ten teams used Cleanroom and five teams used a traditional development approach in a nonparametric design. The major results included: 1) most developers using the Cleanroom approach were able to build systems without program execution; and 2) the Cleanroom teams' products met system requirements more completely and succeeded on more operational test cases than did those developed with a traditional approach. The results suggested the feasibility of complete off-line development, as in Cleanroom, and advanced the understanding of software development methodology.

Other replicated project studies include [37], [4], and [63].

C. Multiproject Variation Studies

With a *motivation* to improve the understanding of resource usage during software development, Bailey and Basili [5] conducted a study whose *purpose* was to predict development cost by using a particular model (i.e., *object*) and to evaluate it from the *perspective* of the project manager, corporate manager, and researcher. The particular model generation method was examined in a multiproject *scope*, with baseline data from 18 large (2500–100 000 source line) software projects in the NASA S.E.L. [27], [26], [38], [91] production environment (from the program *domain*), in which teams contained from two to ten programmers (from the programmer *domain*). The study *design* incorporated multivariate methods to parameterize the model. Objective and subjective *measurement* of the projects was based on 21 *criteria*[2] in three areas: methodology, complexity, and personnel experience. Study *preparation* included preliminary work [52], *execution* included an established set of data collection forms [27], and *analysis* used forward multivariate regression methods. The major results (in the *interpretation context* of the study purpose) included 1) the estimation of software development resource usage improved by considering a set of both baseline and customization factors; 2) the application in the NASA environment of

[2] Twenty-one factors were selected after examining a total of 82 factors that possibly contributed to project resource expenditure, including 36 from [108] and 16 from [28].

the proposed model generation method, which considers both types of factors, produced a resource usage estimate for a future project within one standard deviation of the actual; and 3) the confirmation of the NASA S.E.L. formula that the cost per line of reusing code is 20 percent of that of developing new code. A major result (in the *interpretation context* of the field of research) was that the study highlighted the difference of each software development environment, which improved the selection and use of resource estimation models. The particular programming environment and projects sampled qualify the *extrapolation* of the results. The *impact* of the study was an advancement in the understanding of estimating software development resource expenditure.

In order to assess, manage, and improve multiproject environments, several researchers [28], [20], [108], [10], [36], [21], [62], [112], [97], [107] have characterized, evaluated, and/or predicted the effect of several factors from the perspective of the developer, modifier, project manager, and corporate manager. All the studies examined moderate to large projects from production environments. The relationships investigated were among various factors, including structured programming, personnel background, development process and product constraints, project complexity, human and computer resource consumption, error-prone software identification, error/change distributions, data coupling/binding, project duration, staff size, degree of management control, and productivity. These studies have provided increased project visibility, greater understanding of classes of factors sensitive to project performance, awareness of the need for project measurement, and efforts for standardization of definitions. Analysis has begun on incorporating project variation information into a management tool [9], [14].

In order to improve and better understand the software maintenance process, Vessey and Weber [106] conducted an experiment to evaluate the relationship between the rate of maintenance repair and various product and process metrics from the perspective of the developer, user, and the project manager. A total of 447 small (up to 600 statements) commercial and clerical Cobol programs from one Australian organization and two U.S. organizations were analyzed. The product and process metrics included program complexity, programming style, programmer quality, and number of system releases. The major results were: 1) in the Australian organization, program complexity and programming style significantly affected the maintenance repair rate; and 2) in the U.S. organizations, the number of times a system was released significantly affected the maintenance repair rate.

In order to improve the software maintenance process, Adams [1] evaluated operational faults from the perspective of the user, customer, project manager, and corporate manager. The fault history for nine large production products (e.g., operating system releases or their major components) were empirically modeled. He developed an approach for estimating whether and under what circumstances preventively fixing faults in operational software in the field was appropriate. Preventively fixing faults consisted of installing fixes to faults that had yet to be discovered by particular users, but had been discovered by the vendor or other users. The major result was that for the typical user, corrective service was a reasonable way of dealing with most faults after the code had been in use for a fairly long period of time, while preventively fixing high-rate faults was advantageous during the time immediately following initial release.

In order to assess the effectiveness of the testing process, Bowen [31] evaluated estimations of the number of residual faults in a system from the perspective of the customer, developer, and project manager. The study was based on fault data collected from three large (2000–6000 module) systems developed in the Hughes–Fullerton environment. The study partitioned the faults based on severity and analyzed the differences in estimates of remaining faults according to stage of testing. Insights were gained into relationships between fault detection rates and residual faults.

D. Single Project Studies

With a *motivation* to improve software development methodology, Basili and Turner [22] conducted a study whose *purpose* was to characterize the process (i.e., *object*) of iterative enhancement in conjunction with a top-down, stepwise refinement development approach from the *perspective* of the developer. The development process was examined in a single project *scope*, where the authors, two experienced individuals (from the programmer *domain*), built a 17 000 line compiler (from the program *domain*). The study *design* incorporated descriptive methods to capture system evolution. Objective *measurement* of the system was in several *criteria* areas: size, modularity, local/global data usage, and data binding/coupling [62], [104]. Study *preparation* included language design [23], *execution* incorporated static analysis of system snapshots, and *analysis* used descriptive statistics. The results (in the *interpretation context* of the statistical framework) included: 1) the percentage of global variables decreased over time while the percentage of actual versus possible data couplings across modules increased, suggesting the usage of global data became more appropriate over time; and 2) the number of procedures and functions rose over time while the number of statements per procedure or function decreased, suggesting increased modularity. The major result of the study (in the *interpretation context* of the study purpose) was that the iterative enhancement technique encouraged the development of a software product that had several generally desirable aspects of system structure. A major result (in the *interpretation context* of the field of research) was that the study demonstrated the feasibility of iterative enhancement. The particular programming team and project examined qualify the *extrapolation* of the results. The *impact* of the study was an advancement in the understanding of software development approaches.

In order to improve, better understand, and manage the

software development process, Baker [6] evaluated the effect of applying chief programming teams and structured programming in system development from the perspective of the user, developer, project manager, and corporate manager. The large (83 000 line) system, known as "The New York Times Project," was developed by a team of professionals organized as a chief programmer team, using structured code, top-down design, walk-throughs, and program libraries. Several benefits were identified, including reduced development time and cost, reduced time in system integration, and reduced fault detection in acceptance testing and field use. The results of the study demonstrated the feasibility of the chief programmer team concept and the accompanying methodologies in a production environment.

In order to improve their development environments, several researchers [49], [24], [2], [81], [13] have each conducted single project studies to characterize the errors and changes made during a development project. They examined the development of a moderate to large software project, done by a multiperson team, in a production environment. They analyzed the frequency and distribution of errors during development and their relationship with several factors, including module size, software complexity, developer experience, method of detection and isolation, effort for isolation and correction, phase of entrance into the system and observance, reuse of existing design and code, and role of the requirements document. Such analyses have produced fault categorization schemes and have been useful in understanding and improving a development environment.

In order to better understand and improve the use of the Ada® language, Basili *et al.* [55], [12] examined a ground-support system written in Ada to characterize the use of Ada packages from the perspective of the developer. Four professional programmers developed a project of 10 000 source lines of code. Factors such as how package use affected the ease of system modification and how to measure module change resistance were identified, as well as how these observations related to aspects of development and training. The major results were 1) several measures of Ada programs were developed, and 2) there was an indication that a lot of training will be necessary if we are to expect the facilities of Ada to be properly used.

In order to assess and improve software testing methodology, Basili and Ramsey [15], [88] characterized and evaluated the relationship between system acceptance tests and operational usage from the perspective of the developer, project manager, customer, and researcher. The execution coverage of functionally generated acceptance test cases and a sample of operational usage cases was monitored for a medium-size (10 000 line) software system developed in a production environment. The results calculated that 64 percent of the program statements were executed during system operation and that the acceptance test cases corresponded reasonably well to the operational usage. The results gave insights into the relationships among structural coverage, fault detection, system testing, and system usage.

®Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

V. Problem Areas in Experimentation

The following sections identify several problem areas of experimentation in software engineering. These areas may serve as guidelines in the performance of future studies. After mentioning some overall observations, considerations in each of the areas of experiment definition, planning, operation, and interpretation are discussed.

A. Experimentation Overall

There appears to be no "universal model" or "silver bullet" in software engineering. There are an enormous number of factors that differ across environments, in terms of desired cost/quality goals, methodology, experience, problem domain, constraints, etc. [108], [20], [5], [10], [28]. This results in every software development/maintenance environment being different. Another area of wide variation is the many-to-one (e.g., 10:1) differential in human performance [11], [43], [18]. The particular individuals examined in an empirical study can make an enormous difference. Among other considerations, these variations suggest that metrics need to be validated for a particular environment and a particular person to show that they capture what is intended [11], [21]. Thus, experimental studies should consider the potentially vast differences among environments and among people.

B. Experiment Definition

In the definition of the purpose for the experiment, the formulation of intuitive problems into precisely stated goals is a nontrivial task [17], [25]. Defining the purpose of a study often requires the articulation of what is meant by "software quality." The many interpretations and perceptions of quality [32], [39], [72] highlight the need for considering whose perspective of quality is being examined. Thus, a precise specification of the problem to be investigated is a major step toward its solution.

C. Experiment Planning

Experimental planning should have a horizon beyond a first experiment. Controlled studies may be used to focus on the effect of certain factors, while their results may be confirmed in replications [92], [99], [102], [113], [58], [57], [45], [44], [18] and/or larger case studies [5], [16]. When designing studies, consider that a combination of factors may be effective as a "critical mass," even though the particular factors may be ineffective when treated in isolation [16], [107]. Note that formal designs and the resulting statistical robustness are desirable, but we should not be driven exclusively by the achievement of statistical significance. Common sense must be maintained, which allows us, for example, to experiment just to help develop and refine hypotheses [13], [112]. Thus, the experimental planning process should include a series of experiments for exploration, verification, and application.

D. Experiment Operation

The collection of the required data constitutes the primary result of the study operation phase. The data must be carefully defined, validated, and communicated to ensure their consistent interpretation by all persons associated with the experiment: subjects under observation, experimenters, and literature audience [21]. There have been papers in the literature that do not define their data well enough to enable a comparison of results across many projects and environments. We have often contacted experimenters and discovered that different entities were being measured in different studies. Thus, the experimenter should be cautious about the definition, validation, and communication of data, since they play a fundamental role in the experimental process.

E. Experiment Interpretation

The appropriate presentation of results from experiments contributes to their correct interpretation. Experimental results need to be qualified by the particular samples (e.g., programmers, programs) analyzed [17]. The extrapolation of results from a particular sample must consider the representativeness of the sample to other environments [40], [114], [108], [86], [5], [28]. The visibility of the experimental results in professional forums and the open literature provides valuable feedback and constructive criticism. Thus, the presentation of experimental results should include appropriate qualification and adequate exposure to support their proper interpretation.

VI. Conclusion

Experimentation in software engineering supports the advancement of the field through an iterative learning process. The experimental process has begun to be applied in a multiplicity of environments to study a variety of software technology areas. From the studies presented, it is clear that experimentation has proven effective in providing insights and furthering our domain of knowledge about the software process and product. In fact, there is a learning process in the experimentation approach itself, as has been shown in this paper.

We have described a framework for experimentation to provide a structure for presenting previous studies. We also recommend the framework as a mechanism to facilitate the definition, planning, operation, and interpretation of past and future studies. The problem areas discussed are meant to provide some useful recommendations for the application of the experimental process in software engineering. The experimental framework cannot be used in a vacuum; the framework and the lessons learned complement one another and should be used in a synergistic fashion.

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] J.-L. Albin and R. Ferreol, "Collecte et analyse de mesures de logiciel (Collection and analysis of software data)," *Technique et Science Informatiques*, vol. 1, no. 4, pp. 297–313, 1982 (Rairo ISSN 0752-4072).

[3] A. Avizienis, P. Gunningberg, J. P. J. Kelly, L. Strigini, P. J. Traverse, K. S. Tso, and U. Voges, "The UCLA Dedix system: A distributed testbed for multiple-version software," in *Dig. 15th Int. Symp. Fault-Tolerant Comput.*, Ann Arbor, MI, June 19–21, 1985.

[4] J. W. Bailey, "Teaching Ada: A comparison of two approaches," in *Proc. Washington Ada Symp.*, Washington, DC, 1984.

[5] J. W. Bailey and V. R. Basili, "A meta-model for software development resource expenditures," in *Proc. 5th Int. Conf. Software Eng.*, San Diego, CA, 1981, pp. 107–116.

[6] F. T. Baker, "System quality through structured programming," in *AFIPS Proc. 1972 Fall Joint Comput. Conf.*, vol. 41, 1972, pp. 339–343.

[7] V. R. Basili, *Tutorial on Models and Metrics for Software Management and Engineering.* New York: IEEE Computer Society, 1980.

[8] V. R. Basili and F. T. Baker, "Tutorial of structured programming," in *Proc. 11th IEEE COMPCON*, IEEE Cat. No. 75CH1049-6, 1975.

[9] V. R. Basili and C. Doerflinger, "Monitoring software development through dynamic variables," in *Proc. COMPSAC*, Chicago, IL, 1983.

[10] V. R. Basili and K. Freburger, "Programming measurement and estimation in the software engineering laboratory," *J. Syst. Software*, vol. 2, pp. 47–57, 1981.

[11] 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.

[12] V. R. Basili, E. E. Katz, N. M. Panilio-Yap, C. L. Ramsey, and S. Chang, "A quantitative characterization and evaluation of a software development in Ada," *Computer*, Sept. 1985.

[13] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Commun. ACM*, vol. 27, no. 1, pp. 42–52, Jan. 1984.

[14] V. R. Basili and C. L. Ramsey, "Arrowsmith-P—A prototype expert system for software engineering management," in *Proc. Symp. Expert Systems in Government*, Mclean, VA, Oct. 1985.

[15] V. R. Basili and J. R. Ramsey, "Analyzing the test process using structural coverage," in *Proc. 8th Int. Conf. Software Eng.*, London, Aug. 28–30, 1985, pp. 306–312.

[16] V. R. Basili and R. W. Reiter, "A controlled experiment quantitatively comparing software development approaches," *IEEE Trans. Software Eng.*, vol. SE-7, May 1981.

[17] V. R. Basili and R. W. Selby, "Data collection and analysis in software research and management," *Proc. Amer. Statistical Association and Biometric Society Joint Statistical Meetings*, Philadelphia, PA, August 13–16, 1984.

[18] ——, "Comparing the effectiveness of software testing strategies," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1501, May 1985.

[19] ——, "Four applications of a software data collection and analysis methodology," in *Proc. NATO Advanced Study Institute: The Challenge of Advanced Computing Technology to System Design Methods*, Durham, U. K., July 29–Aug. 10, 1985.

[20] ——, "Calculation and use of an environment's characteristic software metric set," in *Proc. 8th Int. Conf. Software Eng.*, London, Aug. 28–30, 1985, pp. 386–393.

[21] V. R. Basili, R. W. Selby, and T. Y. Phillips, "Metric analysis and data validation across FORTRAN projects," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 652–663, Nov. 1983.

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

[23] ——, *SIMPL-T: A Structured Programming Language.* Geneva, IL: Paladin House, 1976.

[24] V. R. Basili and D. M. Weiss, "Evaluation of a software requirements document by analysis of change data," in *Proc. 5th Int. Conf. Software Eng.*, San Diego, CA, Mar. 9–12, 1981, pp. 314–323.

[25] ——, "A methodology for collecting valid software engineering data*," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 728–738, Nov. 1984.

[26] V. R. Basili and M. V. Zelkowitz, "Analyzing medium-scale software developments," in *Proc. 3rd Int. Conf. Software Eng.*, Atlanta, GA, May 1978, pp. 116–123.

[27] V. R. Basili, M. V. Zelkowitz, F. E. McGarry, R. W. Reiter, Jr., W. F. Truszkowski, and D. L. Weiss, "The software engineering laboratory," Software Eng. Lab., NASA/Goddard Space Flight Center, Greenbelt, MD, Rep. SEL-77-001, May 1977.

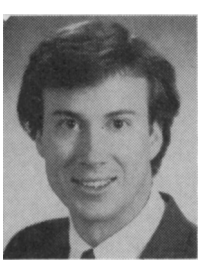
[28] B. W. Boehm, *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[29] 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.
[30] R. C. Bogdan and S. K. Biklen, *Qualitative Research for Education: An Introduction to Theory and Methods.* Boston, MA: Allyn and Bacon, 1982.
[31] J. Bowen, "Estimation of residual faults and testing effectiveness," in *Proc. 7th Minnowbrook Workshop Software Performance Evaluation*, Blue Mountain Lake, NY, July 24–27, 1984.
[32] 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 vols.), Feb. 1985.
[33] G. E. P. Box, W. G. Hunter, and J. S. Hunter, *Statistics for Experimenters.* New York: Wiley, 1978.
[34] F. P. Brooks, Jr., *The Mythical Man-Month.* Reading, MA: Addison-Wesley, 1975.
[35] R. E. Brooks, "Studying programmer behavior: The problem of proper methodology, *Commun. ACM*, vol. 23, no. 4, pp. 207–213, 1980.
[36] W. D. Brooks, "Software technology payoff: Some statistical evidence," *J. Syst. Software*, vol. 2, pp. 3–9, 1981.
[37] F. O. Buck, "Indicators of quality inspections," IBM Systems Products Division, Kingston, NY, Tech. Rep. 21.802, Sept. 1981.
[38] D. N. Card, F. E. McGarry, J. Page, S. Eslinger, and V. R. Basili, "The software engineering laboratory," Software Eng. Lab., NASA/Goddard Space Flight Center, Greenbelt, MD, Rep. SEL-81-104, Feb. 1982.
[39] 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.
[40] W. G. Cochran, *Sampling Techniques.* New York: Wiley, 1953.
[41] W. G. Cochran and G. M. Cox, *Experimental Designs.* New York: Wiley, 1950.
[42] 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.
[43] B. Curtis, "Cognitive science of programming," *6th Minnowbrook Workshop Software Performance Evaluation*, Blue Mountain Lake, NY, July 19–22, 1983.
[44] B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love, "Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics," *IEEE Trans. Software Eng.*, pp. 96–104, Mar. 1979.
[45] B. Curtis, S. B. Sheppard, and P. M. Milliman, "Third time charm: Stronger replication of the ability of software complexity metrics to predict programmer performance," in *Proc. 4th Int. Conf. Software Eng.*, Sept. 1979, pp. 356–360.
[46] M. Dyer, "Cleanroom software development method," IBM Federal Systems Division, Bethesda, MD, Oct. 14, 1982.
[47] M. Dyer and H. D. Mills, "Developing electronic systems with certifiable reliability," in *Proc. NATO Conf.*, Summer 1982.
[48] T. Emerson, "A discriminant metric for module cohesion," in *Proc. 7th Int. Conf. Software Eng.*, Orlando, FL, 1984, pp. 294–303.
[49] A. Endres, "An analysis of errors and their causes in systems programs," *IEEE Trans. Software Eng.*, pp. 140–149, vol. SE-1, June 1975.
[50] A. R. Feuer and E. B. Fowlkes, "Some results from an empirical study of computer software," in *Proc. 4th Int. Conf. Software Eng.*, 1979, pp. 351–355.
[51] R. W. Floyd, "Assigning meaning to programs," *Amer. Math. Soc.*, vol. 19, J. T. Schwartz, Ed., Providence, RI, 1967.
[52] K. Freburger and V. R. Basili, "The software engineering laboratory: Relationship equations," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-764, May 1979.
[53] J. D. Gannon, "An experimental evaluation of data type conventions," *Commun. ACM*, vol. 20, no. 8, pp. 584–595, 1977.
[54] J. D. Gannon and J. J. Horning, "The impact of language design on the production of reliable software," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 179–191, 1975.
[55] J. D. Gannon, E. E. Katz, and V. R. Basili, "Characterizing Ada programs: Packages," in *The Measurement of Computer Software Performance*, Los Alamos Nat. Lab., Aug. 1983.
[56] A. L. Goel, "Software reliability and estimation techniques," Rome Air Development Center, Griffiss Air Force Base, NY, Rep. RADC-TR-82-263, Oct. 1982.
[57] J. D. Gould, "Some psychological evidence on how people debug computer programs," *Int. J. Man-Machine Studies*, vol. 7, pp. 151–182, 1975.
[58] J. D. Gould and P. Drongowski, "An exploratory study of computer program debugging," *Human Factors*, vol. 16, no. 3, pp. 258–277, 1974.
[59] M. H. Halstead, *Elements of Software Science.* New York: North-Holland, 1977.
[60] W. C. Hetzel, "An experimental analysis of program verification methods," Ph.D. dissertation, Univ. North Carolina, Chapel Hill, 1976.
[61] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–583, Oct. 1969.
[62] D. H. Hutchens and V. R. Basili, "System structure analysis: Clustering with data bindings," *IEEE Trans. Software Eng.*, vol. SE-11, Aug. 1985.
[63] S.-S. V. Hwang, "An empirical study in functional testing, structural testing, and code reading/inspection*," Dep. Comput. Sci., Univ. Maryland, College Park, Scholarly Paper 362, Dec. 1981.
[64] Z. Jelinski and P. B. Moranda, "Applications of a probability-based model to a code reading experiment," in *Proc. IEEE Symp. Comput. Software Rel.*, New York, 1973, pp. 78–81.
[65] W. L. Johnson, S. Draper, and E. Soloway, "An effective bug classification scheme must take the programmer into account," in *Proc. Workshop High-Level Debugging*, Palo Alto, CA, 1983.
[66] J. P. J. Kelly, "Specification of fault-tolerant multi-version software: Experimental studies of a design diversity approach," Ph.D. dissertation, Univ. California, Los Angeles, 1982.
[67] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 96–109, Jan. 1986.
[68] R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice.* Reading, MA: Addison-Wesley, 1979.
[69] B. Littlewood, "Stochastic reliability growth: A model for fault renovation computer programs and hardware designs," *IEEE Trans. Rel.*, vol. R-30, Oct. 1981.
[70] B. Littlewood and J. L. Verrall, "A Bayesian reliability growth model for computer software," *Appl. Statist.*, vol. 22, no. 3, 1973.
[71] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. SE-2, pp. 308–320, Dec. 1976.
[72] 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.
[73] R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman, "Program indentation and comprehensibility," *Commun. ACM*, vol. 26, no. 11, pp. 861–867, Nov. 1983.
[74] T. Moher and G. M. Schneider, "Methodology and experimental research in software engineering," *Int. J. Man-Machine Studies*, vol. 16, no. 1, pp. 65–87, 1982.
[75] S. A. Mulaik, *The Foundations of Factor Analysis.* New York: McGraw-Hill, 1972.
[76] J. D. Musa, "A theory of software reliability and its application," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 312–327, 1975.
[77] ——, "Software reliability measurement," *J. Syst. Software*, vol. 1, no. 3, pp. 223–241, 1980.
[78] G. L. Myers, *Composite/Structured Design.* New York: Van Nostrand Reinhold, 1978.
[79] ——, "A controlled experiment in program testing and code walkthroughs/inspections," *Commun. ACM*, pp. 760–768, Sept. 1978.
[80] J. Neter and W. Wasserman, *Applied Linear Statistical Models.* Homewood, IL: Richard D. Irwin, 1974.
[81] T. J. Ostrand and E. J. Weyuker, "Collecting and categorizing software error data in an industrial environment*," *J. Syst. Software*, vol. 4, pp. 289–300, 1983.
[82] D. J. Panzl, "Experience with automatic program testing," in *Proc. NBS Trends and Applications*, Nat. Bureau Standards, Gaithersburg, MD, May 28, 1981, pp. 25–28.
[83] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.
[84] ——, "Some conclusions from an experiment in software engineering techniques," in *AFIPS Proc. 1972 Fall Joint Comput. Conf.*, vol. 41, 1972, pp. 325–329.
[85] ——, "A technique for module specification with examples," *Commun. ACM*, vol. 15, May 1972.
[86] L. Putnam, "A general empirical solution to the macro software sizing and estimating problem," *IEEE Trans. Software Eng.*, vol. SE-4, July 1978.

[87] H. R. Ramsey, M. E. Atwood, and J. R. Van Doren, "Flowcharts versus program design languages: An experimental comparison," *Commun. ACM*, vol. 26, no.6, pp. 445-449, June 1983.
[88] J. Ramsey, "Structural coverage of functional testing," in *Proc. 7th Minnowbrook Workshop Software Perform. Eval.*, Blue Mountain Lake, NY, July 24-27, 1984.
[89] *Statistical Analysis System (SAS) User's Guide*, SAS Inst. Inc., Box 8000, Cary, NC 27511, 1982.
[90] H. Scheffe, *The Analysis of Variance*. New York: Wiley, 1959.
[91] "Annotated bibliography of software engineering laboratory (SEL) literature," Software Eng. Lab., NASA/Goddard Space Flight Center, Greenbelt, MD, Rep. SEL-82-006, Nov. 1982.
[92] R. W. Selby, "An empirical study comparing software testing techniques," in *Proc. 6th Minnowbrook Workshop Software Perform. Eval.*, Blue Mountain Lake, NY, July 19-22, 1983.
[93] ——, "Evaluations of software technologies: Testing, CLEANROOM, and metrics," Ph.D. dissertation, Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1500, 1985.
[94] R. W. Selby, V. R. Basili, and F. T. Baker, "CLEANROOM software development: An empirical evaluation," Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1415, Feb. 1985.
[95] J. G. Shanthikumar, "A statisical time dependent error occurrence rate software reliability model with imperfect debugging," in *Proc. 1981 Nat. Comput. Conf.*, June 1981.
[96] B. A. Sheil, "The psychological study of programming," *Comput. Surveys*, vol. 13, pp. 101-120, Mar. 1981.
[97] V. Y. Shen, T. J. Yu, S. M. Thebaut, and L. R. Paulsen, "Identifying error-prone software—An empirical study," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 317-324, Apr. 1985.
[98] B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems*. Winthrop, 1980.
[99] B. Shneiderman, R. E. Mayer, D. McKay, and P. Heller, "Experimental investigations of the utility of detailed flowcharts in programming," *Commun. ACM*, vol. 20, no. 6, pp. 373-381, 1977.
[100] S. Siegel, *Nonparametric Statistics for the Behavioral Sciences*. New York: McGraw-Hill, 1955.
[101] E. Soloway, J. Bonar, and K. Ehrlich, "Cognitive strategies and looping constructs: An empirical study," *Commun. ACM*, vol. 26, no.11, pp. 853-860, Nov. 1983.
[102] E. Soloway and K. Ehrlich, "Empirical studies of programming knowledge," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 595-609, Sept. 1984.
[103] E. Soloway, K. Ehrlich, J. Bonar, and J. Greenspan, "What do novices know about programming?" in *Directions in Human-Computer Interactions*, A. Badre and B. Shneiderman, Eds. Norwood, NJ: Ablex, 1982.
[104] W. P. Stevens, G. L. Myers, and L. L. Constantine, "Structural design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115-139, 1974.
[105] L. G. Stucki, "New directions in automated tools for improving software quality," in *Current Trends in Programming Methodology*, R. T. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1977.
[106] I. Vessey and R. Weber, "Some factors affecting program repair maintenance: An empirical study," *Commun. ACM*, vol. 26, no. 2, pp. 128-134, Feb. 1983.
[107] J. Vosburgh, B. Curtis, R. Wolverton, B. Albert, H. Malec, S. Hoben, and Y. Liu, "Productivity factors and programming environments," in *Proc. 7th Int. Conf. Software Eng.*, Orlando, FL, 1984, pp. 143-152.
[108] C. E. Walston and C. P. Felix, "A method of programming measurement and estimation," *IBM Syst. J.*, vol. 16, no. 1, pp. 54-73, 1977.
[109] G. Weinberg, *The Psychology of Computer Programming*. New York: Van Nostrand Rheinhold, 1971.
[110] M. Weiser, "Programmers use slices when debugging," *Commun. ACM*, vol. 25, pp. 446-452, July 1982.
[111] M. Weiser and J. Shertz, "Programming problem representation in novice and expert programmers," *Int. J. Man-Machine Studies*, vol. 19, pp. 391-398, 1983.
[112] D. M. Weiss and V. R. Basili, "Evaluating software development by analysis of changes: Some data from the software engineering laboratory," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 157-168, Feb. 1985.
[113] L. Weissman, "Psychological complexity of computer programs: An experimental methodology," *SIGPLAN Notices*, vol. 9, no. 6, pp. 25-36, June 1974.
[114] R. Wolverton, "The cost of developing large scale software," *IEEE Trans. Comput.*, vol. C-23, June 1974.
[115] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen, "The effect of modularization and comments on program comprehension," Dep. Comput. Sci., Arizona State Univ., Tempe, AZ, Working Paper, 1981.
[116] J. C. Zolnowski and D. B. Simmons, "Taking the measure of program complexity," in *Proc. Nat. Comput. Conf.*, 1981, pp. 329-336.

Victor R. Basili (M'83-SM'84) is Professor and Chairman of the Department of Computer Science 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.

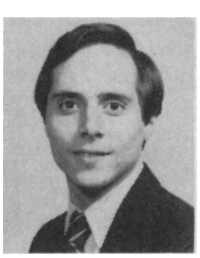
In 1982 Dr. Basili received the Outstanding Paper Award from the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING for his paper on the evaluation of methodologies. He was Program Chairman for the Sixth International Conference on Software Engineering, and the First ACM SIGSOFT Software Engineering Symposium 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.



Richard W. Selby (M'85) was born in Chicago, IL, in 1959. He 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.

Since 1985 he has been 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 metrics. He is currently involved with the development and evaluation of the Arcadia software development environment.

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



David H. Hutchens (M'84) received the B.S. degree in mathematics from Western Carolina University, Cullowhee, NC, in 1977, the M.S. degree in mathematical sciences from Clemson University, Clemson, SC, in 1979, and the Ph.D. degree in computer science from the University of Maryland, College Park, in 1983.

He is currently an Assistant Professor of Computer Science at Clemson University. His research interests include measurement, evaluation, and modeling of the software development process and its product.

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