A Methodology for Collecting Valid Software Engineering Data

VICTOR R. BASILI, MEMBER, IEEE, AND DAVID M. WEISS

Abstract—An effective data collection method for evaluating software development methodologies and for studying the software development process is described. The method uses goal-directed data collection to evaluate methodologies with respect to the claims made for them. Such claims are used as a basis for defining the goals of the data collection, establishing a list of questions of interest to be answered by data analysis, defining a set of data categorization schemes, and designing a data collection form.

The data to be collected are based on the changes made to the software during development, and are obtained when the changes are made. To ensure accuracy of the data, validation is performed concurrently with software development and data collection. Validation is based on interviews with those people supplying the data. Results from using the methodology show that data validation is a necessary part of change data collection. Without it, as much as 50 percent of the data may be erroneous.

Feasibility of the data collection methodology was demonstrated by applying it to five different projects in two different environments. The application showed that the methodology was both feasible and useful.

Index Terms—Data collection, data collection methodology, error analysis, error classification, software engineering experimentation.

I. Introduction

According to the mythology of computer science, the first computer program ever written contained an error. Error detection and error correction are now considered to be the major cost factors in software development [1]-[3]. Much current and recent research is devoted to finding ways of preventing software errors. This research includes areas such as requirements definition [4], automatic and semiautomatic program generation [5], [6], functional specification [7], abstract specification [8]-[11], procedural specification [12], code specification [13]-[15], verification [16]-[18], coding techniques [19]-[24], error detection [25], testing [26], [27], and language design [16], [28]-[31].

One result of this research is that techniques claimed to be effective for preventing errors are in abundance. Unfortunately, there have been few attempts at experimental verification of such claims. The purpose of this paper is to show how to obtain valid data that may be used both to learn more about the software development process and to evaluate software development methodologies in production environments. Previous [15], [32]-[34] and companion [35] papers present data and evaluation results, obtained from two different software development environments. (Not all of the techniques previously mentioned were included in these studies.) The methodology described in this paper was developed as part of studies conducted by the Naval Research Laboratory (NRL) and by NASA's Software Engineering Laboratory (SEL) [36].

The remainder of this section discusses motivation for data collection and the attributes of a useful data collection effort. Section II is a step-by-step description of the data collection methodology. Section III describes the application of the methodology to the SEL environment. Section IV summarizes the lessons learned concerning data collection and its associated problems, limitations, and applications.

Software Engineering Experimentation

The course of action in most sciences when faced with a question of opinion is to obtain experimental verification. Software engineering disputes are infrequently settled that way. Data from experiments exist, but rarely apply to the question to be settled. There are a number of reasons for this state of affairs. Probably the two most important are the number of potential confounding factors involved in software studies and the expense of attempting to do controlled studies in an industrial environment involving medium or large scale systems.

Rather than attempting controlled studies, we have devised a method for conducting accurate causal analyses in production environments. Causal analyses are efforts to discover the causes of errors and the reasons that changes are made to software. Such analyses are designed to provide some insight into the software development and maintenance processes, help confirm or reject claims made for different methodologies, and lead to better techniques for prevention, detection, and correction of errors. Relatively few examples of this kind of study exist in the literature; some examples are [4], [15], [32], [37], [38].

Attributes of Useful Data Collection

To provide useful data, a data collection methodology must display certain attributes. Since much of the data of interest are collected during the test phase, complete analysis of the data must await project completion. For accuracy reasons, it is important that data collection and validation proceed concurrently with development.

Developers can provide data as they make changes during development. In a reasonably well-controlled software development environment, documentation and code are placed under some form of configuration control before being released to

Manuscript received December 13, 1982; revised January 11, 1984. This work was supported in part by the National Aeronautics and Space Administration under Grant NSF-5123 to the University of Maryland.

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

D. M. Weiss is with the Naval Research Laboratory, Washington, DC 20375.

0098-5589/84/1100-0728$01.00 © 1984 IEEE

their users. Changes may then be defined as alterations to baselined design, code, or documentation.

A key factor in the data gathering process is validation of the data as they become available. Such validity checks result in corrections to the data that cannot be made at later times owing to the nature of human memory [39]. Timeliness of both data collection and data validation is quite important to the accuracy of the analysis.

Careful validation means that the data to be collected must be carefully specified, so that those supplying data, those validating data, and those performing the analyses will have a consistent view of the data collected. This is especially important for the purposes of repetition of the studies in both the same and different environments.

Careful specification of the data requires the data collectors to have a clear idea of the goals of the study. Specifying goals is itself an important issue, since, without goals, one runs the risk of collecting unrelated, meaningless data.

To obtain insight into the software development process, the data collectors need to know the kinds of errors committed and the kinds of changes made. To identify troublesome issues, the effort needed to make each change is necessary. For greatest usefulness, one would like to study projects from software production environments involving teams of programmers.

We may summarize the preceding as the following six criteria.

1) The data must contain information permitting identification of the types of errors and changes made.

2) The data must include the cost of making changes.

3) Data to be collected must be defined as a result of clear specification of the goals of the study.

4) Data should include studies of projects from production environments, involving teams of programmers.

5) Data analysis should be historical; data must be collected and validated concurrently with development.

6) Data classification schemes to be used must be carefully specified for the sake of repeatability of the study in the same and different environments.

II. Schema for the Investigative Methodology

Our data collection methodology is goal oriented. It starts with a set of goals to be satisfied, uses these to generate a set of questions to be answered, and then proceeds step-by-step through the design and implementation of a data collection and validation mechanism. Analysis of the data yields answers to the questions of interest, and may also yield a new set of questions. The procedure relies heavily on an interactive data validation process; those supplying the data are interviewed for validation purposes concurrently with the software development process. The methodology has been used in two different environments to study five software projects developed by groups with different backgrounds, using very different software development methodologies. In both environments it yielded answers to most questions of interest and some insight into the development methodologies used. Table I is a summary of characteristics of completed projects that have been studied. Definitions of the characteristics are the same as in [40]. All examples used in this paper are taken from studies of the SEL environement.

TABLE I
Summary of Project Information

	SEL1	SEL2	SEL3	NRL1
Effort (work-months)	79.0	39.6	98.7	48.0
Number of developers	5	4	7	9
Lines of code (K)	50.9	75.4	85.4	21.8
Developed lines of code	46.5	31.1	78.6	21.8
Number of components	502	490	639	253

The projects studied vary widely with respect to factors such as application, size, development team, methodology, hardware, and support software. Nonetheless, the same basic data collection methodology was applicable everywhere. The schema used has six basic steps, listed in the following, with considerable feedback and iteration occurring at several different places.

1) Establish the Goals of the Data Collection: We divide goals into two categories: those that may be used to evaluate a particular software development methodology relative to the claims made for it, and those that are common to all methodologies to be studied.

As an example, a goal of a particular methodology, such as information hiding [41], might be to develop software that is easy to change. The corresponding data collection goal is to evaluate the success of the developers in meeting this goal, i.e., evaluate the ease with which the software can be changed. Goals in this category may be of more interest to those who are involved in developing or testing a particular methodology, and must be defined cooperatively with them.

A goal that is of interest regardless of the methodology being used is to help understand the environment and focus attention on techniques that are useful there. Another such goal is to characterize changes in ways that permit comparisons across projects and environments. Such goals may interest software engineers, programmers, managers, and others more than goals that are specific to the success or failure of a particular methodology.

Consequences of Omitting Goals: Without goals, one is likely to obtain data in which either incomplete patterns or no patterns are discernible. As an example, one goal of an early study [15] was to characterize errors. During data analysis, it became desirable to discover the fraction of errors that were the result of changes made to the software for some reason other than to correct an error. Unfortunately, none of the goals of the study was related to this type of change, and there were no such data available.

2) Develop a List of Questions of Interest: Once the goals of the study have been established, they may be used to develop a list of questions to be answered by the study. Questions of interest define data parameters and categorizations that permit quantitative analysis of the data. In general, each goal will result in the generation of several different questions of interest. As an example, if the goal is to characterize changes, some corresponding questions of interest are: "What is the distribution of changes according to the reason for the change?", "What is the distribution of changes across system components?", "What is the distribution of effort to design changes?"

As a second example, if the goal is to evaluate the ease with which software can be changed, we may identify questions of interest such as: "Is it clear where a change has to be made in the software?", "Are changes confined to single modules?", "What was the average effort involved in making a change?"

Questions of interest form a bridge between subjectively determined goals of the study and the quantitative measures to be used in the study. They permit the investigators to determine the quantities that need to be measured and the aspects of the goals that can be measured. As an example, to discover how a design document is being used, one might collect data that show how the document was being used when the need for a change to it was discovered. This may be the only aspect of the document's use that is measurable.

In addition to forcing sharper definition of goals, questions of interest have the desirable property of forcing the investigators to consider the data analyses to be performed before any data are collected.

Goals for which questions of interest cannot be formulated and goals that cannot be satisfied because adequate measures cannot be defined may be discarded. Once formulated, questions can be evaluated to determine if they completely cover their associated goals and if they define quantitative measures.

Consequences of Omitting Questions of Interest: Without questions of interest, data distributions that are needed for evaluation purposes, such as the distribution of effort involved in making changes, may have to be constructed in an ad hoc way, and be incomplete or inaccurate. As a result, there may be no quantitative basis for satisfying the goals of the study. In effect, goals are not well defined if questions of interest are not or cannot be formulated.

3) Establish Data Categories: Once the questions of interest have been established, categorization schemes for the changes and errors to be examined may be constructed. Each question generally induces a categorization scheme. If one question is, "What was the distribution of changes according to the reason for the change?", one will want to classify changes according to the reason they are made. A simple categorization scheme of this sort is *error corrections* versus *nonerror corrections* (hereafter called *modifications*).

Each of these categories may be further subcategorized according to reason. As an example, modifications could be subdivided into modifications resulting from requirements changes, modifications resulting from a change in the development support environment (e.g., compiler change), planned enhancements, optimizations, and others.

Such a categorization permits characterization of the changes with respect to the stability of the development environment, with respect to different kinds of development activities, etc. When matched with another categorization such as the difficulty of making changes, this scheme also reveals which changes are the most difficult to make.

Each categorization scheme should be complete and consistent, i.e., every change should fit exactly one of the subcategories of the scheme. To ensure completeness, we usually add the category "Other" as a subcategory. Where some changes are not suited to the scheme, the subcategory "Not Applicable" may be used. As an example, if the scheme includes subcategories for different levels of effort in isolating error causes, then errors for which the cause need not be isolated (e.g., clerical errors noticed when reading code) belong in the "Not Applicable" subcategory.

Consequences of Not Defining Data Categories Before Collecting Data: Omitting the data categorization schemes may result in data that cannot later be identified as fitting any particular categorization. Each change then defines its own category, and the result is an overwhelming multiplicity of data categories, with little data in each category.

4) Design and Test Data Collection Form: To provide a permanent copy of the data and to reinforce the programmers' memories, a data collection form is used. Form design was one of the trickiest parts of the studies conducted, primarily because forms represent a compromise among conflicting objectives. Typical conflicts are the desire to collect a complete, detailed set of data that may be used to answer a wide range of questions of interest, and the need to minimize the time and effort involved in supplying the data. Satisfying the former leads to large, detailed forms that require much time to fill out. The latter requires a short, check-off-the-boxes type of form.

Including the data suppliers in the form design process is quite beneficial. Complaints by those who must use the form are resolved early (i.e., before data collection begins), the form may be tailored to the needs of the data suppliers (e.g., for use in configuration management), and the data suppliers feel they are a useful part of the data collection process.

The forms must be constructed so that the data they contain can be used to answer the questions of interest. Several design iterations and test periods are generally needed before a satisfactory design is found.

Our principal goals in form design were to produce a form that

a) fit on one piece of paper,
b) could be used in several different programming environments, and
c) permitted the programmer some flexibility in describing the change.

Fig. 1 shows the last version of the form used for the SEL studies reported here. (An earlier version of the form was significantly modified as a result of experience gained in the data collection and analysis processes.) The first sections of the form request textual descriptions of the change and the reason it was made. Following sections contain questions and check-off tables that reflect various categorization schemes.

As an example, a categorization of time to design changes is requested in the first question following the description of the change. The completer of the form is given the choice of four categories (one hour or less, one hour to one day, one day to three days, and more than three days) that cover all possibilities for design time.

Consequences of Not Using a Data Collection Form: Without a data collection form, it is necessary to rely on the developer's memories and on perusal of early versions of design documentation and code to identify and categorize the changes made. This approach leads to incomplete, inaccurate data.

5) Collect and Validate Data: Data are collected by requiring those people who are making software changes to complete a change report form for each change made, as soon as the change is completed. Validation consists of checking the forms for correctness, consistency, and completeness. As part of the validation process, in cases where such checks reveal problems, the people who filled out the forms are interviewed. Both

NUMBER _____

CHANGE REPORT FORM

PROJECT NAME _____ CURRENT DATE _____

SECTION A - IDENTIFICATION

REASON: Why was the change made? _____

DESCRIPTION: What change was made? _____

EFFECT: What components (or documents) are changed? (Include version) _____

EFFORT: What additional components (or documents) were examined in determining what change was needed? _____

	(Month	Day	Year)
Need for change determined on			
Change started on			

What was the effort in person time required to understand and implement the change?

____1 hour or less, ____1 hour to 1 day, ____1 day to 3 days, ____more than 3 days

SECTION B - TYPE OF CHANGE (How is this change best characterized?)

- ☐ Error correction
- ☐ Planned enhancement
- ☐ Implementation of requirements change
- ☐ Improvement of clarity, maintainability, or documentation
- ☐ Improvement of user services
- ☐ Insertion/deletion of debug code
- ☐ Optimization of time/space/accuracy
- ☐ Adaptation to environment change
- ☐ Other (Explain in E)

Was more than one component affected by the change? Yes _____ No _____

FOR ERROR CORRECTIONS ONLY

SECTION C - TYPE OF ERROR (How is this error best characterized?)

- ☐ Requirements incorrect or misinterpreted
- ☐ Functional specifications incorrect or misinterpreted
- ☐ Design error, involving several components
- ☐ Error in the design or implementation of a single component
- ☐ Misunderstanding of external environment, except language
- ☐ Error in use of programming language/compiler
- ☐ Clerical error
- ☐ Other (Explain in E)

FOR DESIGN OR IMPLEMENTATION ERRORS ONLY

If the error was in design or implementation:

The error was a mistaken assumption about the value or structure of data _____

The error was a mistake in control logic or computation of an expression _____

580-2 (6/78)

(a)

FOR ERROR CORRECTIONS ONLY

SECTION D - VALIDATION AND REPAIR

What activities were used to validate the program, detect the error, and find its cause?

	Activities Used for Program Validation	Activities Successful in Detecting Error Symptoms	Activities Tried to Find Cause	Activities Successful in Finding Cause
Pre-acceptance test runs				
Acceptance testing				
Post-acceptance use				
Inspection of output				
Code reading by programmer				
Code reading by other person				
Talks with other programmers				
Special debug code				
System error messages				
Project specific error messages				
Reading documentation				
Trace				
Dump				
Cross-reference/attribute list				
Proof technique				
Other (Explain in E)				

What was the time used to isolate the cause?

____one hour or less, ____one hour to one day, ____more than one day, ____never found

If never found, was a workaround used? _____Yes _____No (Explain in E)

Was this error related to a previous change?

_____Yes (Change Report #/Date _____) ____No ____Can't tell

When did the error enter the system?

____requirements ____functional specs ____design ____coding and test ____other ____can't tell

SECTION E - ADDITIONAL INFORMATION

Please give any information that may be helpful in categorizing the error or change, and understanding its cause and its ramifications.

Name: _____ Authorized: _____ Date: _____

580-2 (6/78)

(b)

Fig. 1. SEL change report form. (a) Front. (b) Back.

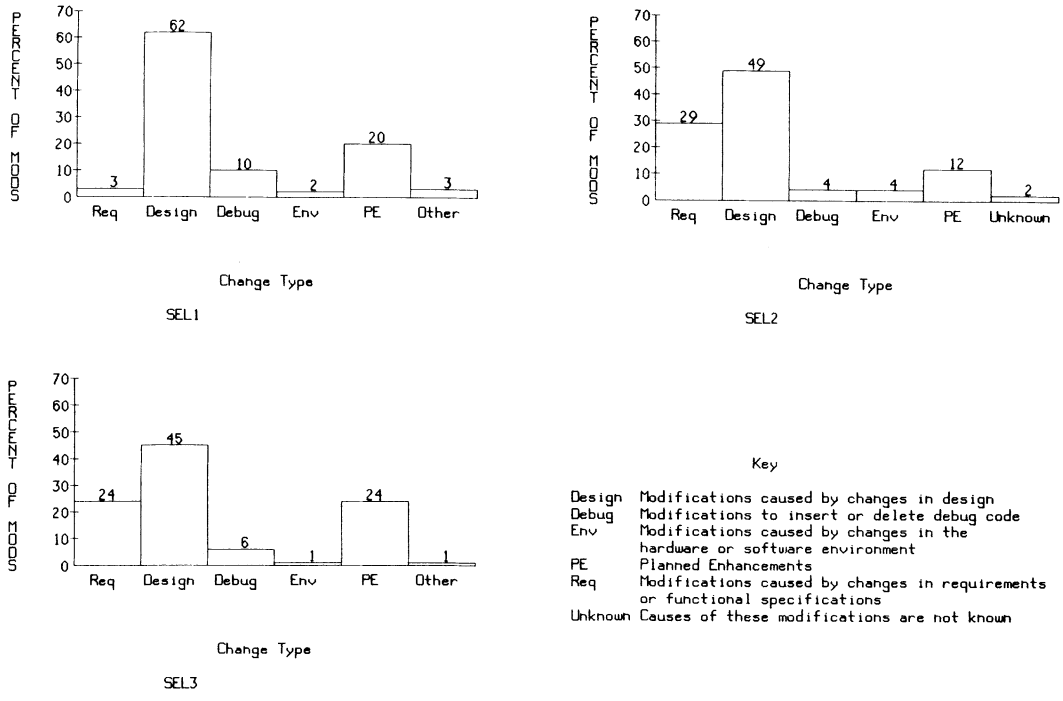


Fig. 2. Sources of modifications.

collection and validation are concurrent with software development; the shorter the lag between completing the form and conducting the interview, the more accurate the data.

Perhaps the most significant problem during data collection and validation is ensuring that the data are complete, i.e., that every change has been described on a form. The better controlled the development process, the easier this is to do. At each stage of the process where configuration control is imposed, change data may be collected. Where projects that we have studied use formal configuration control, we have integrated the configuration control procedures and the data collection procedures, using the same forms for both, and taking advantage of configuration control procedures for validation purposes. Since all changes must be reviewed by a configuration control board in such cases, we are guaranteed capture of all changes, i.e., that our data are complete. Furthermore, the data collection overhead is absorbed into the configuration control overhead, and is not visible as a separate source of irritation to the developers.

Consequences of Omitting Validation: One result of concurrent development, data collection, and data validation is that the accuracy of the collection process may be quantified. Accuracy may be calculated by observing the number of mistakes made in completing data collection forms. One may then compare, for any data category, prevalidation distributions with postvalidation distributions. We call such an analysis a validation analysis. The validation analysis of the SEL data shows that it is possible for inaccuracies on the order of 50 percent to be introduced by omitting validation. To emphasize the consequences of omitting the validation procedures, we present some of the results of the validation analysis of the SEL data in Section III.

6) Analyze Data: Data are analyzed by calculating the parameters and distributions needed to answer the questions of interest. As an example, to answer the question "What was the distribution of changes according to the reason for the change?", a distribution such as that shown in Fig. 2 might be computed from the data.

Application of the Schema

Applying the schema requires iterating among the steps several times. Defining the goals and establishing the questions of interest are tightly coupled, as are establishing the questions of interest designing and testing the form(s), and collecting and validating the data. Many of the considerations involved in implementing and integrating the steps of the schema have been omitted here so that the reader may have an overview of the process. The complete set of goals, questions of interest, and data categorizations for the SEL projects are shown in [33].

Support Procedures and Facilities

In addition to the activities directly involved in the data collection effort, there are a number of support activities and facilities required. Included as support activites are testing the forms, collection and validation procedures, training the programmers, selecting a database system to permit easy analysis of the data, encoding and entering data into the database, and developing analysis programs.

III. Details of SEL Data Collection and Validation

In the SEL environment, program libraries were used to support and control software development. There was a full-time librarian assigned to support SEL projects. All project library changes were routed through the librarian. In general, we define a change to be an alteration to baselined design, code, or documentation. For SEL purposes, only changes to code, and documentation contained in the code, were studied. The

program libraries provided a convenient mechanism for identifying changes.

Each time a programmer caused a library change, he was required to complete a change report form (Fig. 1). The data presented here are drawn from studies of three different SEL projects, denoted SEL1, SEL2, and SEL3. The processing procedures were as follows.

1) Programmers were required to complete change report forms for all changes made to library routines.

2) Programs were kept in the project library during the entire test phase.

3) After a change was made a completed change report form describing the change was submitted. The form was first informally reviewed by the project leader. It was then sent to the SEL library staff to be logged and a unique identifier assigned to it.

4) The change analyst reviewed the form and noted any inconsistencies, omissions, or possible miscategorizations. Any questions the analyst had were resolved in an interview with the programmer. (Occasionally the project leader or system designer was consulted rather than the individual programmer.)

5) The change analyst revised the form as indicated by the results of the programmer interview, and returned it to the library staff for further processing. Revisions often involved cases where several changes were reported on one form. In these cases, the analyst ensured that there was only one change reported per form; this often involved filling out new forms. Forms created in this way are known as *generated* forms. (Changes were considered to be different if they were made for different reasons, if they were the result of different events, or if they were made at substantially different times, e.g., several weeks apart. As an example, two different requirements amendments would result in two different change reports, even if the changes were made at the same time in the same subroutine.) Occasionally, one change was reported on several different forms. The forms were then merged into one form, again to ensure one and only one change per form. Forms created in this way are known as *combined* forms.

6) The library staff encoded the form for entry into the (automated) SEL database. A preliminary, automated check of the form was made via a set of database support programs. This check, mostly syntactic, ensured that the proper kinds of values were encoded into the proper fields, e.g, that an alphabetic character was not entered where an integer was required.

7) The encoded data were entered into the SEL database.

8) The data were analyzed by a set of programs that computed the necessary distributions to answer the questions of interest.

Many of the reported SEL changes were error corrections. We define an error to be a discrepancy between a specification and its implementation. Although it was not always possible to identify the exact location of an error, it was always possible to identify exactly each error correction. As a result, we generally use the term error to mean error correction.

For data validation purposes, the most important parts of the data collection procedure are the review by the change analyst, and the associated programmer interview to resolve uncertainties about the data.

The SEL validation procedures afforded a good chance to discover whether validation was really necessary; it was possible to count the number of miscategorizations of changes and associated misinformation. These counts were obtained by counting the number of times each question on the form was incorrectly answered.

An example is misclassifications of errors as clerical errors. (Clerical errors were defined as errors that occur in the mechanical translation of an item from one format to another, e.g., from one coding sheet to another, or from one medium to another, e.g., coding sheets to cards.) For one of the SEL projects, 46 errors originally classified as clerical were actually errors of other types. (One of these consisted of the programmer forgetting to include several lines of code in a subroutine. Rather than clerical, this was classified as an error in the design or implementation of a single component of the system.) Initially, this project reported 238 changes, so we may say that about 19 percent of the original reports were misclassified as clerical errors.

The SEL validation process was not good for verifying the completeness of the reported data. We cannot tell from the validation studies how many changes were never reported. This weakness can be eliminated by integrating the data collection with stronger configuration control procedures.

Validation Differences Among SEL Projects

As experience was gained in collecting, validating, and analyzing data for the SEL projects, the quality of the data improved significantly, and the validation procedures changed slightly. For SEL1 and SEL2, completed forms were examined and programmers interviewed by a change analyst within a few weeks (typically 3-6 weeks) of the time the forms were completed. For project SEL2, the task leader (lead programmer for the project) examined each form before the change analysts saw it.

Project SEL3 was not monitored as closely as SEL1 and SEL2. The task leader, who was the same as for SEL2, by then understood the data categorization schemes quite well and again examined the forms before sending them to the SEL. The forms themselves were redesigned to be simpler but still capture nearly all the same data. Finally, several of the programmers were the same as on project SEL2 and were experienced in completing the forms.

Estimating Inaccuracies in the Data

Although there is no completely objective way to quantify the inaccuracy in the validated data, we believe it to be no more than 5 percent for SEL1 and SEL2. By this we mean that no more than 5 percent of the changes and errors are misclassified in any of the data collection categories. For the major categories, such as whether a change is an error or modification, the type of change, and the type of error, the inaccuracy is probably no more than 3 percent.

For SEL3, we attempted to quantify the results of the validation procedures more carefully. After validation, forms were categorized according to our confidence in their accuracy. We used four categories.

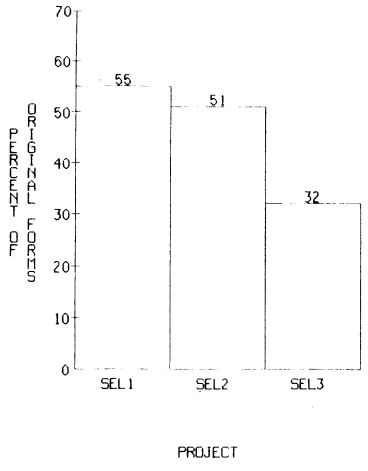


Fig. 3. Corrected forms.

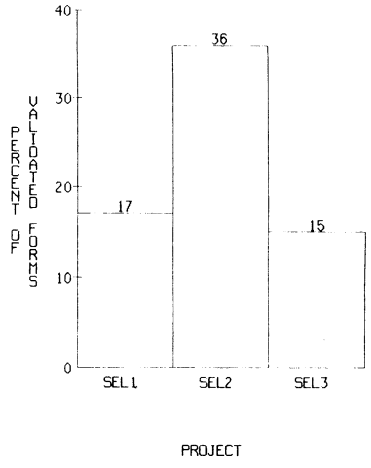


Fig. 4. Generated forms.

1) Those forms for which we had no doubt concerning the accuracy of the data. Forms in this category were estimated to have no more than a 1 percent chance of inaccuracy.

2) Those forms for which there was little doubt about the accuracy of the data. Forms in this category were estimated to have at most a 10 percent chance of an inaccuracy.

3) Those forms for which there was some uncertainty about the accuracy, with an estimated inaccuracy rate of more than 30 percent.

4) Those forms for which there was considerable uncertainty about the accuracy, with an estimated inaccuracy rate of about 50 percent.

Applying the inaccuracy rates to the number of forms in each category gave us an estimated inaccuracy of at most 3 percent in the validated forms for SEL3.

Prevalent Mistakes in Completing Forms

Clear patterns of mistakes and misclassifications in completing forms became evident during validation. As an example, programmers on projects SEL1 and SEL2 frequently included more than one change on one form. Often this was a result of the programmers sending the changes to the library as a group.

Comparative Validation Results

Fig. 3 provides an overview of the results of the validation process for the 3 SEL projects. The percentage of original forms that had to be corrected as a result of the validation process is shown. As an example, 32 percent of the originally completed change report forms for SEL3 were corrected as a result of validation. The percentages are based on the number of original forms reported (since some forms were generated, and some combined, the number of changes reported after validation is different than the number reported before validation). Fig. 4 shows the number of generated forms expressed as a percentage of total validated forms.

Fig. 3 shows that prevalidation SEL3 forms were significantly more accurate than the prevalidation SEL1 or SEL2 forms. Fig. 4 shows that SEL3 also had the lowest incidence of generated forms. Although not shown in the figures, combined forms represented a very small fraction of the total validated forms. Based on this analysis, the prevalidation SEL3 data are considerably better than the prevalidation data for either of the other projects. We believe the reasons for this are the improved design of the form, and the familiarity of the task leader and programmers with the data collection process.

These results show that careful validation, including programmer interviews, is essential to the accuracy of any study involving change data. Furthermore, it appears that with well-designed forms and programmer training, there is improvement with time in the accuracy of the data one can obtain. We do not believe that it will ever be possible to dispense entirely with programmer interviews, however.

Erroneous Classifications

Table II shows misclassifications of errors as modifications and modifications as errors. As an example, for SEL1, 14 percent of the original forms were classified as modifications, but were actually errors. Without the validation process, considerable inaccuracy would have been introduced into the initial categorization of changes as modifications or errors.

Table III is a sampling of other kinds of classification errors that could contribute significantly to inaccuracy in the data. All involve classification of an error into the wrong subcategory. The first row shows errors that were classified by the programmer as clerical, but were later reclassified as a result of the validation process into another category. For SEL1, significant inaccuracy (19 percent) would be introduced by omitting the validation process.

Table IV is similar to Table III, but shows misclassifications involving modifications for SEL1 and SEL3 (SEL2 data were not analyzed for this purpose). The first row shows modifications that were classified by the programmer as requirements or specifications changes, but were reclassified as a result of validation.

Variation in Misclassification

Data on misclassifications of change and error type subcategories, such as shown in Table III, tend to vary considerably among both projects and subcategories. (Misclassification of clerical errors, as shown in Table III, is a good example.) This is most likely because the misclassifications represent biases in the judgments of the programmers. It became clear during the

TABLE II
ERRONEOUS MODIFICATION AND ERROR CLASSIFICATIONS (PERCENT OF ORIGINAL FORMS)

	SEL1	SEL2	SEL3
Modifications classified as errors	1%	5%	less than 1%
Errors classified as modifications	14%	5%	2%

TABLE III
TYPICAL ERROR TYPE MISCLASSIFICATIONS (PERCENT OF ORIGINAL FORMS)

Original Classification	SEL1	SEL2	SEL3
Clerical Error	19%	7%	6%
(Use of) Programming Language	0%	5%	3%
Incorrect or Misinterpreted Requirements	Unavailable	0%	less than 1%
Design Error	Unavailable	8%	1%

TABLE IV
ERRONEOUS MODIFICATION CLASSIFICATIONS (PERCENT OF ORIGINAL FORMS)

	SEL1	SEL3
Requirements or specification change	1%	less than 1%
Design change	8%	1%
Optimization	8%	less than 1%
Other	3%	less than 1%

validation process that certain programmers tended toward particular misclassifications.

The consistency between projects SEL2 and SEL3 in Table III probably occurs because both projects had the same task leader, who screened all forms before sending them to the SEL for validation.

Conclusions Concerning Validation

The preceding sections have shown that the validation process, particularly the programmer interviews, are a necessary part of the data collection methodology. Inaccuracies on the order of 50 percent may be introduced without this form of validation. Furthermore, it appears that with appropriate form design and programmer experience in completing forms, the inaccuracy rate may be substantially reduced, although it is doubtful that it can be reduced to the level where programmer interviews may be omitted from the validation procedures.

A second significant conclusion is that the analysis performed as part of the validation process may be used to guide the data collection project; the analysis results show what data can be reliably and practically collected, and what data cannot be. Data collection goals, questions of interest, and data collection forms may have to be revised accordingly.

IV. RECOMMENDATIONS FOR DATA COLLECTORS

We believe we now have sufficient experience with change data collection to be able to apply it successfully in a wide variety of environments. Although we have been able to make comparisons between the data collected in the two environments we have studied, we would like to make comparisons with a wider variety of environments. Such comparisons will only be possible if more data become available. To encourage the establishment of more data collection projects, we feel it is important to describe a successful data collection methodology, as we have done in the preceding sections, to point out the pitfalls involved, and to suggest ways of avoiding those pitfalls.

Procedural Lessons Learned

Problems encountered in various procedural aspects of the studies were the most difficult to overcome. Perhaps the most important are the following.

1) Clearly understanding the working environment and specifying the data collection procedures were a key part of conducting the investigation. Misunderstanding by the programmer of the circumstances that require him/her to file a change report form will prejudice the entire effort. Prevention of such misunderstandings can in part be accomplished by training procedures and good forms design, but feedback to the development staff, i.e., those filling out the data collection forms, must not be omitted.

2) Similarly, misunderstanding by the change analyst of the circumstances that required a change to be made will result in misclassifications and erroneous analyses. Our SEL data collection was helped by the use of a change analyst who had previously worked in the NASA environment and understood the application and the development procedures used.

3) Timely data validation through interviews with those responsible for reporting errors and changes was vital, especially during the first few projects to use the forms. Without such validation procedures, data will be severely biased, and the developers will not get the feedback to correct the procedures they are using for reporting data.

4) Minimizing the overhead imposed on the people who were required to complete change reports was an important factor in obtaining complete and accurate data. Increased overhead brought increased reluctance to supply and discuss data. In projects where data collection has been integrated with configuration control, the visible data collection and validation overhead is significantly decreased, and is no longer an important factor in obtaining complete data. Because configuration control procedures for the SEL environment were informal, we believe we did not capture all SEL changes.

5) In cases where an automated database is used, data consistency and accuracy checks at or immediately prior to analysis are vital. Errors in encoding data for entry into the database will otherwise bias the data.

Nonprocedural Lessons Learned

In addition to the procedural problems involved in designing and implementing a data collection study, we found several other pitfalls that could have strongly affected our results and their interpretation. They are listed in the following.

1) Perhaps the most significant of these pitfalls was the danger of interpreting the results without attempting to understand factors in the environment that might affect the data. As an example, we found a surprisingly small percentage of interface errors on all of the SEL projects. This was surprising since interfaces are an often-cited source of errors. There was also other evidence in the data that the software was quite amenable to change. In trying to understand these results, we discussed them with the principal designer of the SEL projects (all of which had the same application). It was clear from the discussion that as a result of their experience with the application,

the designers had learned what changes to expect to their systems, organized the design so that the expected changes would be easy to make, and then reused the design from one project to the next. Rather than misinterpreting the data to mean that interfaces were not a significant software problem, we were led to a better understanding of the environment we were studying.

2) A second pitfall was underestimating the resources needed to validate and analyze the data. Understanding the change reports well enough to conduct meaningful, efficient programmer interviews for validation purposes initially consumed considerable amounts of the change analysts' time. Verifying that the database was internally consistent, complete, and consistent with the paper copies of reports was a continuing source of frustration and a sink for time and effort.

3) A third potential pitfall in data collection is the sensitivity of the data. Programmers and designers sometimes need to be convinced that error data will not be used against them. This did not seem to be a significant problem on the projects studied for a variety of reasons, including management support, processing of the error data by people independent of the project, identifying error reports in the analysis process by number rather than name, informing newly hired project personnel that completion of error reports was considered part of their job, and high project morale. Furthermore, project management did not need error data to evaluate performance.

4) One problem for which there is no simple solution is the Hawthorne (or observer) effect [42]. When project personnel become aware that an aspect of their behavior is being monitored, their behavior will change. If error monitoring is a continuous, long-term activity that is part of the normal scheme of software development, not associated with evaluation of programmer performance, this effect may become insignificant. We believe this was the case with the projects studied.

5) The sensitivity of error data is enhanced in an environment where development is done on contract. Contractors may feel that such data are proprietary. Rules for data collection may have to be contractually specified.

Avoiding Data Collection Pitfalls

In the foregoing sections a number of potential pitfalls in the data collection process have been described. The following list includes suggestions that help avoid some of these pitfalls.

1) Select change analysts who are familiar with the environment, application, project, and development team.

2) Establish the goals of the data collection methodology and define the questions of interest before attempting any data collection. Establishing goals and defining questions shoud be an iterative process performed in concert with the developers. The developers' interests are then served as well as the data collector's.

3) For initial data collection efforts, keep the set of data collection goals small. Both the volume of data and the time consumed in gathering, validating, and analyzing it will be unexpectedly large.

4) Design the data collection form so that it may be used for configuration control, so that it is tailored to the project(s) being studied, so that the data may be used for comparison purposes, and so that those filling out the forms understand the terminology used. Conduct training sessions in filling out forms for newcomers.

5) Integrate data collection and validation procedures into the configuration control process. Data completeness and accuracy are thereby improved, data collection is unobtrusive, and collection and validation become a part of the normal development procedures. In cases where configuration control is not used or is informal, allocate considerable time to programmer interviews, and, if possible, documentation search and code reading.

6) Automate as much of the data analysis process as possible.

Limitations

It has been previously noted that the main limitation of using a goal-directed data collection approach in a production software environment is the inability to isolate the effects of single factors. For a variety of reasons, controlled experiments that may be used to test hypotheses concerning the effects of single factors do not seem practical. Neither can one expect to use the change data from goal-directed data collection to test such hypotheses.

A second major limitation is that lost data cannot be accurately recaptured. The data collected as a result of these studies represent five years of data collection. During that time there was considerable and continuing consideration given to the appropriate goals and questions of interest. Nonetheless, as data were analyzed, it became clear that there was information that was never requested but that would have been useful. An example is the length of time each error remained in the system. Programmers correcting their own errors, which was the usual case, can supply these data easily at the time they correct the error. Our attempts to discover error entry and removal times after the end of development were fruitless. (Error entry times were particularly difficult to discover.) This type of example underscores the need for careful planning prior to the start of data collection.

Recommendations that May Be Provided to the Software Developer

The nature of the data collection methodology and its target environments do not generally permit isolation of the effects of particular factors on the software development process. The results cannot be used to prove that a particular factor in the development process causes particular kinds of errors, but can be used to suggest that certain approaches, when applied in the environment studied, will improve the development process. The software developer may then be provided with a set of recommended approaches for improving the software development process in his environment.

As an example, in the SEL environment neither external problems, such as requirements changes, nor global problems, such as interface design and specification, were significant. Furthermore, the development environment was quite stable. Most problems were associated with the individual programmer. The data show that in the SEL environment it would

clearly pay to impose more control on the process of composing individual routines.

Conclusions Concerning Data Collection for Methodology Evaluation Purposes

The data collection schema presented has been applied in two different environments. We have been able to draw the following conclusions as a result.

1) In all cases, it has been possible to collect data concurrently with the software development process in a software production environment.

2) Data collection may be used to evaluate the application of a particular software development methodology, or simply to learn more about the software development process. In the former case, the better defined the methodology, the more precisely the goals of the data collection may be stated.

3) The better controlled the development process, the more accurate and complete the data.

4) For all projects studied, it has been necessary to validate the data, including interviews with the project developers.

5) As patterns are discerned in the data collected, new questions of interest emerge. These questions may not be answerable with the available data, and may require establishing new goals and questions of interest.

Motivations for Conducting Similar Studies

The difficulties involved in conducting large-scale controlled software engineering experiments have as yet prevented evaluations of software development methodologies in situations where they are often claimed to work best. As a result, software engineers must depend on less formal techniques that can be used in real working environments to establish long-term trends. We view goal-oriented data collection as one such technique and feel that more techniques, and many more results obtained by applying such techniques, are needed.

ACKNOWLEDGMENT

The authors thank the many people at NASA/GSFC and Computer Sciences Corporation who filled out forms and submitted to interviews, especially J. Grondalski and Dr. G. Page, and the librarians, especially S. DePriest.

We thank Dr. J. Gannon, Dr. R. Meltzer, F. McGarry, Dr. G. Page, Dr. D. Parnas, Dr. J. Shore, and Dr. M. Zelkowitz for their many helpful suggestions.

Deserving of special mention is F. McGarry, who had sufficient foresight and confidence to sponsor much of this work and to offer his projects for study.

REFERENCES

[1] B. Boehm *et al.*, *Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980's (CCIP-85)*, Space and Missile Syst. Org., Los Angeles, CA, Feb. 1972.

[2] B. Boehm, "Software and its impact: A quantitative assessment," *Datamation*, vol. 19, pp. 48–59, May 1973.

[3] R. Wolverton, "The cost of developing large scale software," *IEEE Trans. Comput.*, vol. C-23, no. 6, 1974.

[4] T. Bell, D. Bixler, and M. Dyer, "An extendable approach to computer-aided software requirements engineering," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 49–60, Jan. 1977.

[5] A. Ambler, D. Good, J. Browne, *et al.*, "GYPSY: A language for specification and implementation of verifiable programs," in *Proc. ACM Conf. Language Design for Reliable Software*, Mar. 1977, pp. 1–10.

[6] Z. Manna and R. Waldinger, "Synthesis: Dreams ⇒ programs," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 294–329, July 1979.

[7] K. Heninger, "Specifying requirements for complex systems: New techniques and their application," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 2–13, Jan. 1980.

[8] D. L. Parnas, "A technique for software module specification wtih examples," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 330–336, May 1972.

[9] J. Guttag, "The specification and application to programming of abstract data types," Comput. Syst. Res. Group, Dep. Comput. Sci., Univ. Toronto, Ont., Canada, Rep. CSRG-59, 1975.

[10] —, "Abstract data types and the development of data structures," *Commun. Ass. Comput. Mach.*, vol. 20, pp. 396–404, June 1976.

[11] B. Liskov and S. Zilles, "Specification techniques for data abstraction," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 7–19, Mar. 1975.

[12] H. Mills, R. Linger, and B. Witt, *Structured Programming Theory and Practice*. Reading, MA: Addison-Wesley, 1979.

[13] S. Caine and E. Gordon, "PDL–A tool for software design," in *Proc. Nat. Comput. Conf.*, 1975, pp. 271–276.

[14] H. Elovitz, "An experiment in software engineering: The architecture research facility as a case study," in *Proc. 4th Int. Conf. Software Eng.*, 1979, pp. 145–152.

[15] D. Weiss, "Evaluating software development by error analysis: The data from the architecture research facility," *J. Syst. Software*, vol. 1, pp. 57–70, 1979.

[16] E. W. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1976.

[17] R. W. Floyd, "Assigning meanings to programs," in *Proc. XIX Symp. Appl. Math.*, Amer. Math. Soc., 1967, pp. 19–32.

[18] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. Ass. Comput. Mach.*, vol. 12, pp. 576–580, Oct. 1969.

[19] F. Baker, "Chief programmer team management of production programming," *IBM Syst. J.*, vol. 11, no. 1, pp. 56–73, 1972.

[20] E. W. Dijkstra, "Notes on structured programming," in *Structured Programming*. London, England: Academic, 1972.

[21] D. E. Knuth, "Structured programming with go to statements," *Comput. Surveys*, vol. 6, pp. 261–301, Dec. 1974.

[22] H. Mills, "Chief programmer teams: Principles and procedures," IBM Fed. Syst. Div., FSC 71-5108, 1971.

[23] —, "Mathematical foundations for structured programming," IBM Fed. Syst. Div., FSC 72-6012, 1972.

[24] N. Wirth, "Program development by stepwise refinement," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 221–227, Apr. 1971.

[25] E. Satterthwaite, "Debugging tools for high-level languages," *Software–Practice and Exp.*, vol. 2, pp. 197–217, July–Sept. 1972.

[26] W. Howden, "Theoretical and empirical studies of program testing," in *Proc. 3rd Int. Conf. Software Eng.*, May 1978, pp. 305–310.

[27] J. Goodenough and S. Gerhart, "Toward a theory of test data selection," in *Proc. Int. Conf. Reliable Software*, 1975, pp. 493–510.

[28] J. Gannon, "Language design to enhance programming reliability," Comput. Syst. Res. Group, Dep. Comput. Sci., Univ. Toronto, Toronto, Ont., Canada, Rep. CSRG-47, 1975.

[29] J. Gannon and J. Horning, "Language design for programming reliability," *IEEE Trans. Software Eng.*, vol. SE-1, June 1975.

[30] C. A. R. Hoare and N. Wirth, "An axiomatic definition of the programming language Pascal," *Acta Inform.*, vol. 2, pp. 335–355, 1973.

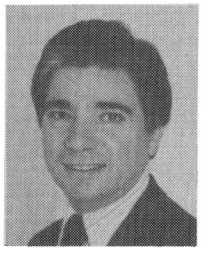
[31] K. Jensen and N. Wirth, *Pascal User Manual and Report*, 2nd ed. New York: Springer-Verlag, 1974.

[32] V. Basili and D. Weiss, "Evaluation of a software requirements document by analysis of change data," in *Proc. 5th Int. Conf. Software Eng.*, Mar. 1981, pp. 314–323.

[33] D. Weiss, "Evaluating software development by analysis of change data," Comput. Sci. Cen., Univ. Maryland, College Park, Rep. TR-1120, Nov. 1981.

[34] L. Chmura and D. Weiss, "Evaluation of the A-7E software requirements document by analysis of changes: Three years of data," presented at *NATO AGARD Avionics Symp.*, Sept. 1982.

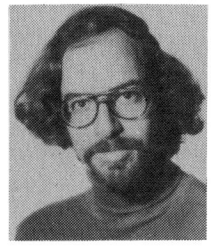
[35] V. Basili and D. Weiss, "Evaluating software development by analysis of changes: Some data from the Software Engineering Laboratory," *IEEE Trans. Software Eng.*, to be published.
[36] V. Basili, M. Zelkowitz, F. McGarry, *et al.*, "The Software Engineering Laboratory," Univ. Maryland, College Park, Rep. TR-535, May 1977.
[37] B. Boehm, "An experiment in small-scale application software engineering," TRW, Rep. TRW-SS-80-01, 1980.
[38] A. Endres, "Analysis and causes of errors in systems programs," in *Proc. Int. Conf. Reliable Software*, 1975, pp. 327–336.
[39] G. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," *Psychol. Rev.*, vol. 63, pp. 81–97, Mar. 1956.
[40] J. Bailey and V. Basili, "A meta-model for software development resource expenditures," in *Proc. 5th Int. Conf. Software Eng.*, Mar. 1981, pp. 107–116.
[41] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 1053–1058, Dec. 1972.
[42] J. Brown, *The Social Psychology of Industry*. Baltimore, MD: Penguin, 1954.



Victor R. Basili (M'83) received the Ph.D. degree in computer science from the University of Texas at Austin.

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

Dr. Basili is a member of the Association for Computing Machinery and the IEEE Computer Society. He has been Program Chairman for several conferences and has served on several editorial boards.



David M. Weiss received the B.S. degree in mathematics in 1964 from Union College and the M.S. and Ph.D. degrees in computer science from the University of Maryland, College Park, in 1974 and 1981, respectively.

Since 1975 he has been on the research staff at the Naval Research Laboratory, Washington, DC, currently with the Computer Science and Systems Branch. His research interests are in software engineering, software change analysis, and formal specification. He is a member of the software cost reduction project whose purpose is to provide a well-engineered model of a complex real-time system.

A Concurrent General Purpose Operator Interface

NEIL B. CORRIGAN AND J. DENBIGH STARKEY

Abstract—**Compact interactive control consoles are replacing traditional control rooms as operator interfaces for physical processes. In the first major application of concurrent programming outside the area of operating systems, this paper presents a design for a general purpose operator interface which uses a color graphics terminal with a touch-sensitive screen as the control console. Operators interact with a process through a collection of application-dependent displays generated interactively by users familiar with the physical process. The use of concurrent programming results in a straightforward and reliable design which may easily be extended to support multiple devices of varying types in the control console. An implementation of the Operator Interface in Concurrent Pascal currently in progress is also discussed.**

Manuscript received November 30, 1982; revised November 23, 1983.

N. B. Corrigan was with the Department of Computer Science, Washington State University, Pullman, WA 99164. He is now with Westinghouse Hanford Company, Richland, WA 99352.

J. D. Starkey was with the Department of Computer Science, Washington State University, Pullman, WA 99164. He is now with the Department of Computer Science, Montana State University, Bozeman, MT 59717.

Index Terms—**Computer graphics, Concurrent Pascal, concurrent programming, interactive system, man-machine interface, operator interface, process control.**

I. Introduction

An operator interface for the control of a physical process is designed to provide an operator with information about the current state of the process and allow the operator to send commands to control the process. Existing operator interfaces take on a variety of forms, ranging from the traditional hard-wired control room to the more recently developed compact control station connected to a computer system. Carrying the latter concept further, this paper presents the design of a general purpose operator interface, which represents the first major application of concurrent programming to an area other than operating systems.

In a conventional control room, operators interact with the

0098/5589/84/1100-0738$01.00 © 1984 IEEE