Information Processing & Management Vol. 20, No. 1-2, pp. 269-276, 1984
Printed in Great Britain.

0306-4573/84 $3.00 + .00
Pergamon Press Ltd.

DATA COLLECTION AND EVALUATION FOR EXPERIMENTAL COMPUTER SCIENCE RESEARCH

MARVIN V. ZELKOWITZ
Department of Computer Science, University of Maryland, College Park, MD 20742, U.S.A.

Abstract—The Software Engineering Laboratory has been monitoring software development at NASA Goddard Space Flight Center since 1976. This report describes the data collection activities of the Laboratory and some of the difficulties of obtaining reliable data. In addition, the application of this data collection process to a current prototyping experiment is reviewed.

1. INTRODUCTION

There is a significant need to collect reliable data on software development projects in order to provide an empirical basis for making conclusions about software development methodologies, models and tools. However, such data is usually hard to collect and even harder to evaluate. Software is a multibillion dollar industry where 100% cost overruns are common, and maintenance activities can take up to 70% of the total cost of the system [11]. The availability of reliable data to evaluate competing software development techniques is crucial.

As Lord Kelvin stated, "I often say that when you can measure what you are speaking about, and express it in numbers, you can know something about it, but when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind". The lack of adequate measures is certainly a problem in the software industry today.

Many of the recent analyses of the software development process are based on data that is obtained from university experiments. Students often program special problems whose results are subjected to analysis. This gives the researcher the 10 to 100 data points necessary for statistical validity of the results. However, by virtue of being part of an academic program, such experiments are necessarily small and usually involve inexperienced programmers. There is a need to extend the scope of these experiments to a level appropriate to the multibillion dollar industry.

Most software development data in industry has been collected after the fact. That is, a project is built and then a pile of documents are handed to a research group for evaluation. Often, critical information is missing and the results are not what one would expect. Rather than following the model of archeology—the study of dead software projects, software evaluation must model sociology—the study of living software societies. Data must be collected from ongoing projects, but the software sociologists must not impact the objects of their study. Given the need to finish projects on time and within budgets—a goal too often missed—it is difficult to justify spending money on data collection and evaluation activities.

Specifically to address these problems, the Software Engineering Laboratory (SEL) was set up within NASA Goddard Space FlightCenter in 1976. The goal was to study software development activities within NASA and report on experiences that will improve the process. This report describes the SEL and its experiences over the last six years.

2. THE SOFTWARE ENGINEERING LABORATORY

In 1976 the SEL was organized to study software development within the NASA environment. More specifically, its primary charter was to monitor the development of ground support software for unmanned spacecraft. Each such system was typically 30,000 to 50,000 source lines of Fortran and took from 8 to 10 programmers up to two years to build. While this environment is not representative of all software development environments, SEL experiences are generalizable in some respects:

(a) Ground support software includes several program types such as data base functions, real time processing, scientific calculations and control language functions. The software is largely implemented in Fortran.

(b) By looking at a relatively narrow environment, data collected from many projects can be compared. Thus we get some of the benefits of a carefully controlled experiment without the expense of duplicating large developments. We do not have the problem of looking at a variety of projects, like compilers, COBOL programs, ground support software, MIS programs and then trying to say something consistent about all of these.

To date, 46 projects have been studied, containing over 1.8 million lines of code. Over 150 programmers participated in these projects, and the data base contains over 40 million bytes of data. The general SEL strategy is to carefully monitor a project and regularly collect data during its development. The data is then entered in the SEL data base for analysis. The purpose of this report is not to dwell on specific research results based on this data (See, for example [8] for a collection of published papers about the SEL) but is concerned with the problems of collecting data, and what we have learned from this process.

3. DATA COLLECTION

3.1. *Model generation*

In order to fully take advantage of the available data, it must be known what information is desired. The models and measures that are to be investigated must be defined. A random data collection activity will usually miss relevant data, and then it will be too late to try and recover that information.

In the SEL, two classes of measures were identified for study, and the data collection activities were oriented around those areas. The initial activities included:

(a) *Process measures*. Evaluating personnel and computer resources over time was a clear need. One activity was to try and validate models that others have identified (e.g. the Putnam Norden Rayleigh curve [1]) while another activity was to try and build new models to fit the empirical data (e.g. the Parr curve [7]). Once models were identified, their predictive nature was studied as a means of resource scheduling.

The generation and correction of errors is another activity that has important economic consequences. However, few models are available to build upon, so there was a need to develop new models of errors and investigate their effects upon performance.

(b) *Product measures*. The size, structure, and complexity of software are other important economic factors to consider. The evaluation of measures such as the software science measures of Halstead [5], the cyclomatic complexity of McCabe [6] and other measures developed within the SEL was another early goal.

Reliability is a critical activity in most environments. In our particular environment, the software that was previously developed was highly reliable (typically under 10 errors in an operational system), so that reliability, while important, was not a primary driving force in organizing the SEL.

3.2. *Forms generation*

The first process in evaluating empirical data is the data collection activity. Ideally, you would like the process to be automated and transparent to the programmer. However, this was not possible in this situation. We were interested in the human activities of software development. Thus we needed detailed information about how programmers spend their time. Because of this, a decision made early in the life of the SEL was that some data would be manually collected using a series of forms.

There is a significant tradeoff consideration at this point. If we tried to collect too much information, programmers would object to the interference of the data collection activity on their work. If too little information was asked, then there would be little point in collecting it.

We first developed an initial set of reporting forms. These have been revised several times since then. Each time certain fields were clarified and the amount of information sought decreased somewhat. At the present time, the effort required to fill out the forms is not significant. Initially seven forms were developed. However, only three are used heavily. These seven forms are:

(a) *Resource summary.* This form lists the number of hours per week spent by all personnel on the project. This information is obtained mostly from the weekly time cards supplied by the contractor. It is easy to obtain this data, and causes little overhead to a project. However, it is very useful for monitoring global resource expenditures, especially in conjunction with the following Component Status Report.

(b) *Component status report.* This form is submitted weekly by each programmer. It lists for each component of the system (e.g. Fortran subroutine) the number of hours spent on each of nine categories (e.g. design, code, test, review, etc.). The detail required by this form initially caused some concern; however, in looking over past forms the average programmer worked on only 5–10 components per week and only 2 or 3 activities per component. Thus the overhead was not excessive. While the data is only approximate to the nearest hour, we believe that it is more accurate than many other data collection procedures.

For example, many research papers give percentages for design, code, and test on a project. However, these are usually taken from resource summary data and calendar date milesones. If a design review occurs on a Friday, then all activities up until that date are design, with all activities the next week being code. In the SEL environment, there was approximately a 25 percent error in using calendar dates for percent effort [4]. On four projects, approximately 25 percent of the design occurred during the coding phase, while almost half of the testing occurred prior to the testing phase (Fig. 1). The Component Status Report is critical for a proper view of development activities.

(c) *Change report form.* This form is completed after each change to a component is compleated and tested. Due to the number of changes that a component undergoes during early development, there was no attempt to capture this data before the component was "complete" (i.e. through unit test). Note that we are capturing "changes" and not simply "errors." All modifications, due to errors or other considerations such as enhancements, are tracked.

Besides identifying the type of change, this form also identifies the cause of the change—they are not always the same, although programmers have difficulty separating the two. The form also asks for information on the time to find and correct an error, and what tools and techniques were used in the process.

PROJECT	BY DATE			BY PHASE		
A	22.7	49.6	27.6	30.7	44.7	24.5
B	22.2	68.2	9.5	34.1	45.6	20.2
C	27.4	61.6	11.0	36.8	48.7	14.5
E	30.2	52.3	17.4	42.0	50.4	7.6

(a) Per cent design, code and test by milestone date and actual task

	%DESIGN DURING CODE	%CODE DURING TEST	%TEST DURING DESIGN & CODE
A	23	27	49
B	38	4	67
C	25	8	56
E	25	21	24

(b) Per cent effort during another phase

(Data collection began after the design phase of project D, so it is omitted here.)

Fig. 1. Task breakdown by phase and date.

In some enviornments, the introduction of this form might cause programmers to object; however, this was not the case in our environment. A standard change monitoring procedure was in place, so we simply changed the form that this branch of NASA GSFC was using before the SEL was created.

These three forms provide the most important data collected by the SEL. Four other forms have been created and used with limited success. These are:

(d) *Component summary*. This form identifies the characteristics of each component in a system. It gives the size, complexity and interfaces. The goal was to have this form filled out at least twice—once when the component was first identified during design, and again when it was completed. Our experience was that the initial form was filled out before much relevant information was known, and the data on the final form could be extracted automatically from the source code data base.

(e) *Computer run analysis*. An entry on this form is filled out for each computer run giving characteristics of the run (execution time, purpose of run, components processed) as well as whether the run met its objectives. This is one form that could be automated. However, the usual range of operating system "Completion Codes" is inadequate for many purposes. For example, a debugging run that was expected to fail at a certain statement, but ran to a successful exit, would have a statisfactory completion code, yet it was a failure as a run since the desired error did not occur.

An interactive job submittal system could help. Before any run, the system could prompt for some of this information. After the run, the system could ask what happened. Since the current NASA environment consists primarily of interactive editing with batch processing, such an online process would have been difficult to implement.

(f) *Programmer analyst survey*. This form attempts to characterize the experiences of the programmers on the project in order to get a general profile of the project tea. However, we immediately ran into confidentiality problems concerning personnel records. We never got the detailed information that we desired, but have obtained general comments on each programmer—although the goal is NOT to rate programmers. If there is any hint of any of this data being used for any sort of personnel action, then compliance drops sharply and the value of the data becomes open to question.

(g) *General project summary*. This is a form that provides a high-level description of a project. Since the software is developed by NASA and contractor personnel, the form is somewhat superfluous and the information is entered directly into the data base.

An important consideration in forms development is consistency in collecting data. Along with each form a detailed instruction sheet was developed, as well as a glossary of relevant terms like "component," "line of code," and "life cycle phase." For example, we chose the name "component" rather than "subroutine" or "module" simply because those terms were well known (with alternative meanings) and we did not want to evoke any preconceived but wrong image in the minds of the participants. Even so, there was a great deal of confusion about the meanings of the various terms. During the early days of the SEL, many meetings were held to explain the process to programmers. Since each programmer worked about one year on a project, after six years there is a large core of personnel experienced in filling out our reporting forms.

3.3 *Data processing*

After being filled out, each form is entered into a data base on a PDP 11/70 computer. In addition to the forms previously described, analyzers were run over the source programs to extract additional information, including lines of code and other measures such as the Halstead software science measures.

Another step in forms processing is data validation. Someone must review the forms as they are submitted. This is expensive, but necessary. It is a quick way to catch and correct errors. In addition, the data entry program should check for data consistency and value ranges. For example, if the program is to read in input in the formal MMDDYY, then a month input that is not a number in the range from 01 to 12 must be rejected. A field requiring an input of *A*, *B* or *C* should reject any other value. Even though we manually check each form, a validation program was more effective for catching errors.

All forms, especially the change report form, need to be reviewed by SEL personnel. Two common errors in the Change report form are to turn in one change report form which actually represented several errors, and the submission of multiple forms for the same error. From earlier work over half of the change report forms were modified following a careful study of each form. This is an expensive process, but needs to be done in order to have accurate data about your environment.

Redundancy of data is another important consideration. Collecting the same or similar data on multiple forms allows for cross validation. There ahould be a reasonable correlation between the collected values. The resource summary and component status reports have been easiest to validate. The Computer Run Analysis form is important for validating some of the change report data; however, limited availability of this form has handicapped some of this validation work. Because of that, it is important to manually check each change report form for selected projects.

4. RESEARCH ACTIVITIES

4.1 *Previous research*

Research in the SEL has centered on resource and error models and on predicting software productivity. ([8] is a collection of relevant papers published over the last few years.) Perhaps the most important conclusion—although obvious in hindsight—which is relevant to this current discussion is that there is no typical software development environment.

All models include parameters—factors which represent variables in that environment (Fig. 2 represents a list of factors from the SEL as well as two other studies [3, 10]). When models based on other environments are applied to the NASA environment, they invariably fail. Does that mean that NASA is different? unique? much better or much worse than other environments? For example, SEL programmers show much higher productivity in lines of code per week than in other organizations. Does that mean that other organizations should pirate away NASA's staff?

Perhaps, but another explanation becomes apparent when NASA's environment is studied in detail. In the SEL, most of the projects are similar ground support software systems. Thus the top level design for these projects are similar. Programmers are experts at this particular problem—thus high productivity. Many factors affecting requirements and design do not apply here. On the other hand, a contractor that bids on a variety of projects—an operating system, a compiler, a data base management system, an attitude orbit determination program, etc. does not build an institutional knowledge about any one particular environment. Requirements and design factors now become significant in this environment and productivity drops.

All companies operate in a different manner. Company policy as to working conditions, computer usage (batch or interactive), leave policy and salaries, management, support tools, etc. all effect productivity. Thus each organization (probably even separate divisions within a single organization) has a different structure and a different set of parameters.

For this reason, one must first calibrate any model to be applied. First develop a quantitative relationship using many factors. Choose those factors relevant to your environment. Calibrate the equations based upon previous projects, and then use the calibrated model for prediction [2]. It is this important calibration step that is missing from most models.

For example, if a baseline equation is given by:

$$\text{Effort} = a * \text{size} + b$$

then one can fit a and $b$from historical data; and the units of size can be determined from those relevant to your environment—such as lines of code, lines of source (including comments), number of modules, number of output statements, etc. Thus instead of a single model, there is a class of models tailored to each environment.

Walston and Felix:

Customer experience
Customer participation in definition
Customer interface complexity
Development location
Percent programmers in design
Programmer qualifications
Programmer experience with machine
Programmer experience with language
Programmer experience with application
Worked together on same type of problem
Customer originated program design changes
Hardware under development
Development environment closed
Development environment open with request
Development environment open
Development enviroment RJE
Development environment TSO
Percent code structured
Percent code used code review
Percent code used top-down
Percent code by chief-programmer teams
Complexity of application processing
Complexity of prorgan flow
Complexity of internal communication
Complexity of external communication
Complexity of data-base structure
Percent code non-math and I/O
Percent code math and computational
Percent code CPU and I/O control
Percent code fallback and recovery
Percent code other
Proportion code real time of interactive
Design constraints: main storage
Design constraints: timing
Design constraints: I/O capability
Unclassified

Boehm:

Required fault freedom
Data base size
Product complexity
Adaptation from existing software
Execution time constraint
Main storage constraint
Virtual machine volatility
Computer response time
Analyst capability
Applications experience
Programmer Capability
Virtual machine experience
Programming language experience
Modern programming practices
Use of software tools
Required development Schedule

SEL:

Program design language (development and design)
Formal design review
Tree charts
Design formalisms
Design/decision notes
Walk-through: design
Walk-through: code
Code reading
Tod-down design
Top-down code
Structured code
Librarian
Chief Programmer Teams
Formal Training
Formal test plans
Unit development folders
Formal documentation
Heavy management involvement and control
Iterative enhancement
Individual decisions
Timely specs and no changes
Team size
On schedule
TSO development
Overall
Reusable code
Percent programmer effort
Percent management effort
Amount documentation
Staff size

Fig. 2. Environment factors.

2.2 *Prototypes*

Over the past few years various methodologies have been studied by the SEL. A current SEL activity is the development of software prototypes. Currently software is designed, built and delivered. Rarely is the product evaluated in advance. However, the use of engineering prototypes in a preliminary evaluation is starting to be discussed by software engineering professionals [9].

While the term is appearing with increasing frequency, what does it really mean? Is it a quick and dirty throw-a-way implementation or a carefully designed subset of a final implementation? What are the cost and reliability parameters for a prototype compared to a full implementation.

Currently data on the subject is meagre and usually based on small projects [12]. The SEL is now investigating a larger implementation with some techniques as applied to previous SEL projects.

Briefly, the target implementation is an integrated support system for flight dynamics research. Currently, experiments (NASA scientists), in trying a new spacecraft model (e.g. a new orbit calculation) must understand the structure of the existing system, access the Fortran source modules, modify them, rebuild the operating program, test it, and then run the experiment—a complex and costly process. The new system is expected to "understand" several flight dynamics systems and to provide a higher level command language that guides the experimenter through the process of building a new version of a system, even if the experimenter is not thoroughly familiar with the existing system. This system is basically a command language interpreter with a complex data dictionary describing the underlying flight dynamics subsystems.

This program is quite different from existing software produced by NASA, so the plan is to prototype it first. Two classes of data will be obtained from the prototype.

(a) *Characteristics of the process.* The Computer Science world has little information available about prototyping, thus this data will add to the general knowledge about this process. What does the life cycle of a prototype look like? How much time is spent in design? code? test? Are errors crucial or can they be side-stepped in the prototype somewhat by "eliminating" the offending feature in the requirements?

Similarly, how does prototyping effect the later full implementation? Will design be easier? Will productivity be higher? Will the overall cost of the system plus prototype be less than the cost of just the full system? Will reliability be higher or the interface more "user friendly?"

(b) *Predictive nature of the prototype.* Once a prototype is built, is it successful? How does one measure success? Will the full system be successful based upon an evaluation of the prototype? A set of measures will be built into the prototype to provide some of these answers.

A baseline study will be made of how experiments are conducted—the cost of machine and people resources will be measured. Some of these experiments will be repeated with the prototype to derive a cost. These will be used to predict the cost of using the full system. If acceptable, then the design will be used for the full implementation, if not, then the design will be modified to correct the problem in the full implementation.

In addition, data will be collected on how often features are used in the prototype, and also how often the prototype is being circumsized in order to provide features that currently do not exist but are needed by the users.

Once the final system is built, the predictive model can be validated in order to aid in developing a theory of software prototypes.

CONCLUSIONS

The Software Engineering Laboratory has been in existence for six years and has studied over 40 projects. The empirical data that has been collected supports several conclusions:

(1) Data collection is hard and expensive. It must be dynamically collected during the development of a project and not after completion.

(2) Data must be validated. Error rates on manually filled out forms are high. A lack of standardized nomenclature for the field hurts consistency. Much effort must go in training personnel to understand the data collection methodology.

(3) Each software development environment is unique. Baseline equations must first be calibrated with past projects before a model can be used in the future.

(4) Little is known, but much is being said, about software prototypes. The SEL is currently studying this issue as part of its ongoing activities.

Acknowledgements—This paper was supported by NASA grant NSG-5123 to the University of Maryland. The SEL is under the direction of Frank McGarry of NASA GSFC. Dr. Victor Basili directs the University of Maryland activities of the SEL, and Jerry Page is the coordinator for Computer Sciences Corporation. The results described in this report were developed by the author, the above mentioned persons, as well as several graduate students at the University of Maryland and researchers at CSC. The author is indebted to David Card of CSC for his detailed comments on an earlier draft of this paper.

REFERENCES

[1] V. R. Basili and M. V. Zelkowitz, Analyzing medium scale software developments. *Proc. 3rd Int. Conf. on Software Engineering*, Atlanta, Georgia, May (1978).

[2] V. R. Basili, Models and metrics for software management and engineering. *ASME Advances in Computer Technology* 1, January, (1980).

[3] B. BOEHM, *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, New Jersey (1981).
[4] E. CHEN and M. V. ZELKOWITZ, Use of cluster analysis to evaluate software engineering methodologies. *Proc. 5th Int. Conf. on Software Enginerring*, San Diego California, March (1981).
[5] M. HALSTEAD, *Elements of Software Science*, American Elsevier, New York (1977).
[6] T. MCCABE, A complexity measure. *IEEE Trans. Software Engng* 1976, **2**, 308–320.
[7] F. PARR, An alternative to the Rayleigh Curve model for software development. *IEEE Trans. Software Engng* 1980, **6**, 291–296.
[8] *Collected Software Engineering Papers*, Vol. 1, SEL-82-004, Code 582.1, NASA GSFC, July (1982).
[9] *ACM SIGSOFT Software Engineering Symp.*, Workshop on Rapid Prototyping, Columbia, Maryland, April (1982).
[10] C. WALSTON and C. FELIX, A method of programming measurement and estimation, *IBM Systems J.* 1977, **16**, 54–73.
[11] M. V. ZELKOWITZ, A. C. SHAW and J. D. GANNON, *Principles of Software Engineering and Design*. Prentice Hall, Englewood Cliffs, New Jersey (1979).
[12] M. V. ZELKOWITZ, A case study in rapid prototyping, *Software Pra. Exp.* 1980, **10**, 1037–1042.