# Chapter 8:
# Data Collection, Validation and Analysis

**Victor R. Basili**

Department of Computer Science
University of Maryland
College Park, MD 20742

## Introduction

One of the major problems with doing measurement of the software development process and the product is the ability to collect reliable data that can be used to understand and evaluate the development process and product and the various models and metrics. The data collection process consists of several phases — establishing the environment in which the project is being developed, the actual data collection process itself, the validation of the collection process and the data, and, finally, the careful analysis and interpretation of that data with respect to specific models and metrics. We will discuss each of these phases.

## Establishing the Environment

Before we begin collecting data, we must understand the various factors that affect software development. Data collection should begin with listing those factors one hopes to control, measure, and understand. In this way, we may characterize the environment, understand what we are studying, and be able to isolate the effects. One possible approach is to create categories of factors.

A partial list of factors is given below, categorized by their association with the problem, the people, the process, the product, the resources, and the tools. Some factors may fit in more than one category, but are listed only once.

### People Factors

These include all the individuals involved in the software development process, including managers, analysts, designers, programmers, and librarians. People related factors that can affect the development process include: number of people involved, level of expertise of the individual members, organization of the group, previous experience with the problem, previous experience with the methodology, previous experience with working with other members of the group, ability to communicate, morale of the individuals, and capability of each individual.

### Problem Factors

The problem is the application or task for which a software system is being developed. Problem related factors include: type of problem (mathematical, database manipulation, etc.), relative newness to state-of-the-art requirements, magnitude of the problem, susceptibility to change, new start or modification of an existing system, final product required (e.g., object code, source, documentation, etc.), state of the problem definition (e.g., rough requirements vs. formal specification), importance of the problem, and constraints placed on the solution.

### Process Factors

The process consists of the particular methodologies, techniques, and standards used in each area of the software development. Process factors include: programming languages, process design language, specification language, use of librarian, walk-throughs, test plan, code reading, top down design, top down development (stubs), iterative enhancement, chief programmer team, Nassi-Shneiderman charts, HIPO charts, data flow diagrams, reporting mechanisms, structured programming, and milestones.

### Product Factors

The product of a software development effort is the software system itself. Product factors include: deliverables, size in lines of code, words of memory, etc., efficiency tests, real-time requirements, correctness, portability, structure of control, in-line documentation, structure of data, number of modules, size of modules, connectivity of modules, target machine architecture, and overlay sizes.

### Resource Factors

The resources are the nonhuman elements allocated and expanded to accomplish the software development. Resource factors include: target machine system, development machine system, development software, deadlines, budget, response times, and turnaround times. (Note there is a relationship between resource and product factors in that the resources define a set of limits within which the product must perform. Sometimes these external constraints can be a dominating force on the product and sometimes they are only a minor factor, e.g., it is easy to get the product to perform well within the set of constraints.)

### Tool Factors

The tools, although also a resource factor, are listed separately because of the important impact they have on development. Tools are the various supportive automated aids used during the various phases of the development process. Tool factors include: requirements analyzers, system design analyzers, source code analyzers (e.g., FACES), database systems, PDL processors, automatic flowcharters, automated development libraries, implementation languages, analysis facilities, testing tools, and maintenance tools.

## Collecting the Data

Once it is clear what the environmental factors are, it is important that what data is needed be carefully considered. The data needed should be driven by the basic models and metrics that will be used and studied. However, since this may not always be known beforehand, especially in a research environment, we must also include a second level set of data that involves what we may want to know, model, or measure. Data collected in this bottom-up manner can be used to refine and modify the existing models and metrics and be used to help characterize our environment.

The actual collection process can take four basic formats: reporting forms, interviews, automatic collection using the computer system, and automated data analysis routines. The reporting forms are usually filled out by the various members of the development team from senior management to clerical support. The benefit of participants' filling out the forms is that they can usually give detailed insights into what is really happening on the project and provide great detail in the data. Questions on a form can be much more specific than the kind of information one can collect automatically. On the other hand, automated data collection has the advantage of being more accurate since it is not as subject to human errors. It can also be done without the participants' being aware of what specific activities and factors are being studied.

Form development is an art all by itself. First one needs to know what data is needed. This must be modified by what data the participants would be willing and able to answer accurately. One large factor here is sampling rate, that is, how often can the forms be filled out so that the participant is willing to do it and still remembers what it is you want to know. It is important that a certain amount of redundancy be built into the data collection process so that reliability checks can be made across the data forms.

Before forms are filled out, the participants should be given a training course in filling out the forms. They should be supplied with a glossary of terms, instructions on filling out the forms, and some sample filled out forms. It would also be helpful if the training session covered some of the models the data collectors had in mind so that the participants had a better idea of the kind of information that was wanted. One representative set of forms [1] may look as follows:

### A General Project Summary

This form would be used to classify the project and will be used in conjunction with the other reporting forms to measure the estimated versus actual development progress. It should be filled out by the project manager at the beginning of the project, at each major milestone, and at the end. The final report should accurately describe the system development life cycle.

### A Programmer/Analyst Survey

This form would classify the background of the personnel on each project. It should be filled out once at the start of the project by all personnel.

### A Component Summary

This form would be used to keep track of the components of a system. A component is a piece of the system identified by name or common function (e.g., an entry in a tree chart or baseline diagram for the system at any point in time, or a shared section of data such as a common clock). With the information on this form combined with the information on the Component Status Report, the structure and status of the system and its development can be monitored. This form is filled out for each component at any point in time when a major modification is made. It should be filled out by the person responsible for that component.

### A Component Status Report

This form would be used to keep track of the development of each component in the system. The form is turned in at the end of each week and lists the number of hours spent on each component. This form is filled out by persons working on the project.

### A Resource Summary

This form keeps track of the project costs on a weekly basis. It is filled out by the project manager every week of the project duration. It should correlate closely with the component status report.

### A Change Report Form

The change report form is filled out every time the system changes because of change or error in design, code, specifications, or requirements. The form identifies the error, its cause and other facets of the project that are affected.

### Computer Program Run Analysis

This form is used to monitor the computer activities used in the project. An entry is made every time the computer is used by the person initiating the run.

Interviews are used to validate the accuracy of the forms and to supplement the information contained on them in areas where it is impossible to expect reasonably accurate information in a form format. In the first case, spot check interviews are conducted with individuals filling out the forms to check that they have given correct information as interpreted by an independent observer. This would include agreement about such things as the cause of an error or at what point in the development process the error was caused or detected.

In the second case, interviews can be held to gather information in depth on several management decisions, e.g., why a particular personnel organization was chosen, or why a particular set of people was picked.

These are the kinds of questions that often require discussion rather than a simple answer on a form.

The easiest and most accurate way to gather information is through an automated system. Throughout the history of the project, more and more emphasis should be placed on the automatic collection of data as we become more aware what data we want to collect, i.e., what data is the most valuable and what data we can or need to get, etc. More effort is required in the development or procurement of automatic collection tools.

The most basic information gathering device is the program development library. The librarian can automatically record data and alleviate the clerical burden for the manager and the programmers. Copies of the current state of affairs of the development library can be periodically archived to preserve the history of the developing product.

A second technique for gathering data automatically is to analyze the product itself, gathering information about its structure by using a program analyzer system. What data is gathered depends upon the particular product metrics.

The above data collected on the project should be stored in a computerized database. Data analysis routines can be written to collect derived data from raw data in the database.

The data collection process is clearly iterative. The more we learn, the better informed we are about what other data we need and how better to collect it.

## Data Validation

After archiving, the next stage is to apply validation techniques to the encoded data. The first step in the validation process is a review of the forms as they were handed in by someone connected with the data collection process to make sure that all the forms have been handed in and that the appropriate fields have been filled out. The next step is to enter the data into the database through a program that checks the validity of the data format and rejects data which is outside of the appropriate ranges. For example, this program can assure that all dates are legal dates and that system component names and programmer names are valid for the project. The program does this by using a prestored list of component names and programmer names.

Ideally, all data in the database should be reviewed by individuals who know what the data should look like. Clearly, this is expensive and not always possible. However, several projects should be reviewed for errors in detail and counts of the number of errors and types of errors kept so that error bounds can be calculated for the unchecked data. This allows data to be interpreted with the appropriate care.

Another type of validity check is to examine the consistency of the database by examining redundant data. This can be done by comparing similar data from different sources to assure the data is reasonably accurate. For example, if effort data is collected at the budget level (resource summary data) and at the individual programmer level (component status data), there should be a reasonable correlation between the two total efforts. Another approach is to use cluster analysis to look for patterns of behavior that are indicative of errors in filling out the forms. For example, if all the change report forms filled out by a particular programmer fall into one cluster, it may imply that there is a bias in the data based upon the particular programmer.

It is clear that data collection is a serious problem, especially in the collection of data on large programming projects across many environments where one set of forms may not be enough to capture what is happening in each of the environments. Unfortunately, if we are to compare projects, we do need common data and we need to know how valid that data is in each case so as not to draw improper conclusions.

## Data Analysis and Result Reporting

After the environment has been established, the appropriate data collected and validated, the process of data analysis can begin. The first step entails fitting the data to the specific models and metrics and the interpretation of the results. If the data supports the model, then it reinforces our understanding of the software development process and product. If the data does not support the model, then we must further analyze the model and its application to the data and the data collection environment. It is possible that the data collection environment did not satisfy some of the assumptions of the model, explicit or otherwise. We can use this data to refine or refute the model or to gain new insights into our software development environment. In any case, the application of the model to the data often generates more questions than it answers and sets the stage for new analysis and new data to be collected.

The data analysis process can be motivated by the different needs for understanding. When linked with various models and metrics, the data analysis can be used to evaluate the software development process and product, to help with software development, and to monitor the stability and quality of an existing product. The process of collecting and analyzing data varies with each area of interest.

Better understanding of the software development process and the software development product is a critical need. Metrics can help in that understanding by allowing us to compare different products and different development environments and providing us with insights regarding their characteristics. Too often we think of all software as the same. Metrics can be used to delineate the various software products and environments.

Many metrics have as a major goal the evaluation of the quality of the process or product in a quality assurance environment. Thus a low score on a metric like the number of errors, indicates something desirable about the quality of the process while a high score on the same metric indicates something quite undesirable about the product. Here data can be analyzed after the project is over.

A second use of metrics would be as a tool for development. In this case, the metric can act as feedback to the developer, letting him know how the development is progressing. It can be used to predict where the project is going by estimating future size or cost, or it may tell him his current design is too complicated and unstructured. Metrics should certainly be used across the entire life cycle and as early as possible to facilitate estimation as well as evaluation. Here data must be analyzed in real time and reports generated in a form easily understandable by the software developer.

A third use of metrics is to monitor the stability and quality of the product through maintenance and enhancement; that is, we can periodically recalculate a set of metrics to see if the product has changed character in some way. It can provide a much needed feedback during the maintenance period. If we find over a period of time that more and more control decisions have entered the system, then something may have to be done to counteract this change in character.

This last use of metrics is relativistic, requiring only a simple partial ordering to indicate what is changed. A relative measure is clearly easier to validate than an absolute measure. The first two uses of metrics — the

evaluation of t
predominantly
within the sam
values of the
metric is that w
what is good
somewhere be

Data collect
One must kn
environment.
assumptions a
the results for
environments
factor may acc

When repor
factors as the
statistical res
reporting resu
is a large
measurement.
the measure
comments are
not only exec
figures could
results of an
bounds, not ju
process itself.

There is a g
More work mu
developed wh
development
there is a sing
process. Mo
analysis and
we must be re
report our res
obtained to su
will gain the c

evaluation of the process and product and the tool of development—are predominantly absolute metrics; that is, there is no basis of comparison within the same project. You may only compare their values with the values of the metrics on other projects. The drawback to an absolute metric is that we need some normalization and calibration factor to tell us what is good and what is bad. The data analysis environment here is somewhere between the two discussed above.

Data collected from any project must be interpreted with great care. One must know the nature of the project and its development environment. To use any model or metric, one must fully understand its assumptions as well as its strengths and weaknesses in order to interpret the results for the particular environment. One must generalize to other environments very cautiously and with great reserve. One unmeasured factor may account for a complete change in effect.

When reporting data, one should report the raw data, the various factors as they are understood, and, in the case of experiments, any statistical results independent of interpretation. It is important in reporting results to define the terms used as precisely as possible. There is a large communication problem due to imprecise units of measurement. For example, if size is reported in lines of source code, the measure is dependent upon the language used, whether or not comments are counted and the commenting convention, and whether or not only executable statements are counted. The difference in the figures could be of the order of two or three to one. Whenever the results of an analysis are reported, it is important to publish error bounds, not just in the fit to the model, but in the actual data collection process itself.

There is a great deal of work to be done in the data collection process. More work must be done in defining terms. A variety of models must be developed which provide us with different viewpoints of the software development process, and we must not fall into the trap of assuming that there is a single overall model of software and the software development process. Most important, because of the nature of experimentational analysis and the many factors that contribute to software development, we must be ready to duplicate the studies and experiments of others and report our results in the open literature. It is only when a wealth of data is obtained to support a particular hypotheses that the software community will gain the confidence to believe in it.

## References

[1] Victor R. Basili, Marvin V. Zelkowitz, Frank E. McGarry, Robert W. Reiter, Walter F. Truszkowski, David L. Weiss. *The software engineering laboratory.* Technical Report TR-535, May 1977, University of Maryland, Computer Science Center, College Park, Maryland 20742.

### Questions Asked at the Conference

#### Lorraine Duvall, IIT Research

We're under contract to the Rome Air Development Center to establish what is called a data and analysis center for software (DACS). One of our tasks is to establish a software experience data base with the kind of field data that Bill Curtis discussed. Now we do have in our data base the data from the NASA Software Engineering Laboratory, and the failure interval data from John Musa's reliability work at Bell Laboratories. In the last couple of months we have had a really aggressive data acquisition program, and from a preliminary look at what kind of data is really available out there, we could be swamped in six months. Now, the kinds of data we are dealing with now have been collected as part of a programming support environment. We have got some conversion data that has been collected through the Navy. Now my question is, not only to you but to any of the other members of the panel, if this kind of data base is available where you may have reports that discuss the collection of the data and the raw data itself, is this good enough to actually help you in your research efforts, or do you need more information to really make this data usable to you?

#### Vic Basili

What I would need to know if I were to use that data is whether I really understood what the numbers represented, how good the data was, and what the error bounds were. That is the only way I would feel secure in making use of the data. The data we have collected in the Software Engineering Laboratory I understand well since I was involved in its collection. I know what data is accurate, I know what data is missing, and I know what data is not very reliable. This way, when I test a model or metric or make a prediction, I understand how to interpret the data; that is, whether it is the model or the data which is in error.

arry, Robert

Maryland,
nd 20742.

er to establish
ACS). One of
ith the kind of
data base the
nd the failure
boratories. In
gressive data
kind of data is
iths. Now, the
ed as part of a
onversion data
ion is, not only
iis kind of data
s the collection
o actually help
nation to really

whether I really
e data was, and
d feel secure in
n the Software
involved in its
data is missing,
n I test a model
erpret the data;
r.

I would be very hesitant to use someone else's data unless I had a real feeling for the collection and validation process as well as error bounds, and I would expect that someone else would be concerned about using data collected in the Software Engineering Laboratory at NASA, since they did not have enough feeling on the accuracy bounds. We are trying to validate the data, but as yet there is not enough information on error bounds available. In fact, I have a challenge for you. One thing that DACS could do is not just collect the data, but perform error analysis. For example, they could do redundancy analysis or cluster analysis to find error patterns in the data. This way, when you report the results you could also report about the consistency and error range of the data base.

**Jean Sammet**

Let me also respond to that. I believe that one of the standard problems in attempting to use heterogeneous data bases and data that has come from several sources is that the same information may be encoded in very different ways. The definition of terms, as Vic mentioned, is not consistent and the kind of data that is collected is irregular. Are you at DACS attempting to do anything to unify that? Are you, for example, trying to set up standards for consistency in data collection? I, personally, think that's a contribution that your project could make.

**Lorraine Duvall**

About two years ago we attempted to define a generic data base, trying to model the software development process, and realized that this was no easy task. The approach we're taking now is to work very closely with the IEEE group to define terms. We're looking to the IEEE task group for data collection terminology. However, we do have some data now and we can't wait for three years before we make it into one beautiful data base. So we're approaching it from two different directions. I think that definition of a generic data base for software experience data is really an interesting project and if anybody has the money, we'd really love to do it.

**Merv Muller**

As a statistician, one loves to see data, but I really feel it is a question of what society can afford versus what are the needs of society, and I really think that you ought to know why you want to have the data. In one sense, you want data for descriptive purposes, because this is how you get insight, and evolve, but I don't really believe that that's why we want metrics. I believe that we want it because we want to predict future events. In answer to the question of what kind of data you need, I think there's some kind of priority ranking based on what the important questions are that you're trying to find improved, prediction methods. I don't know what the priorities are, but I would think that some part of the money should be spent on trying to figure out where the real issues are.

**Vic Basili**

I agree one hundred percent with what you just said. I have to understand why I am collecting the data. Although another valid purpose of data collection is to understand how we do business, we must still start with a model, not with data. What is it I want, what is it I want to be able to understand, or predict, or whatever?

**Wayne Bennett, National Bureau of Standards**

Analogous to the efforts that I see here is the computer performance evaluation effort that's been going on for a long time. What they've learned, it seems to me, is that they've collected an inordinate amount of data: test data was providing them with data, accounting logs were providing them with data, and they've finally come to the conclusion that they don't know what they're collecting the data for. They've realized that without a purpose as a touchstone, that there is no reason to throw out this and keep that. The computer performance evaluation users' group is now in its sixteenth year and they've just now begun addressing these questions.

At the risk of sounding ignorant, I cannot understand, short of its impact on programming, what our interest in languages is. I can see, as usual, industry and academia passing like ships in the night. There's a question that still remains and I think it is something that could be explained to those of us that aren't as familiar with some of the reasons for data collecting. I would think that cost, either in computer performance evaluation or in software metrics, is the bottom line. If I have a measure of anything and I can't relate it ultimately to cost, pump it into some simple model that managers will understand, why am I getting that measure?

It could be that the most useful thing that the academic environment could do would be to help decide what those touchstones should be and formally define them. I guess the question that I'm asking is do you see

that as you
opposed tc
reasons fo
concerning

**Jean Sam**

Well, I'm n
consider a
element. (
amount of
dialect and
PASCAL-lik
Now, if I wr
has certair
whether the
may need c
cost eleme
contract, I
know wheth

**Wayne Be**

My questior
end result.
the cost of
that they te
be that the
willing to a
hypotheses,
gets past th
interesting
that the diff
once you c
doesn't mak
believe that
that possibil
forces first.

that as your role? Do you see that as part of what you're trying to do as opposed to simply collecting data? Do you actually want to put forth the reasons for data collecting? I would ask this also of Ms. Sammet concerning her first slide on high level language metrics.

**Jean Sammet**

Well, I'm not about to repeat the whole talk which I'm sure you would consider a blessing. But let me give you one example of the cost element. One of the things we don't know how to do is measure the amount of deviation of one language from another. We use the term dialect and we use the term ALGOL-like language or PL/I-like language, PASCAL-like language, but we don't know what those things mean. Now, if I write a contract with somebody to produce me a language that has certain characteristics, it would be nice to be able to measure whether the terms of that contract had been fulfilled. In order to do that I may need certain measurements with regard to the languages. That's a cost element which doesn't relate to programming but relates to the contract, I asked to have some product created for me and I'd like to know whether or not it really has been.

**Wayne Bennett**

My question is whether or not it's not really important to ask what is the end result. If the gentleman in the back (Capers Jones) is correct and the cost of documentation requirements and specification are so large that they tend to swamp the choice of language question, then it could be that there is an initial limit on what kinds of questions we're even willing to ask about choice of languages. Those are the kinds of hypotheses, it seems to me, that should be tested first. And after one gets past those, if indeed you get past those, then you can ask the more interesting questions about classification of languages. If it turns out that the difference is a one percent difference in the end in the cost, once you count in this man's documentation cost, it could be that it doesn't make a lick of difference and it's not worth looking into. I don't believe that that's wholly true; however, it seems to me that because of that possibility one has to start to look at it from the top, at the driving forces first.

**Vic Basili**

Let me answer that question by coming back to data collection. Data collection has to be driven by the models or the metrics I'm interested in, and clearly cost is going to be one of them. It may be the prime one in the end. I can talk about cost by looking at cost models so that I will be able to predict future costs. But I'm also interested in studying other factors, such as complexity metrics. It may be important to understand whether it is going to be hard to develop software metrics. I believe or hypothesize that there's a direct correlation between complexity measures and the cost of maintenance. The more complex the program is going to be the more expensive it's going to be to maintain. So, my end goal is to minimize my costs. You're probably right, the goal of everything is cost.

**Jean Sammet**

The problem is the one of cost and we ought to solve it and delineate what the issues are before we tackle anything else. Fortunately or unfortunately, the world is very unhomogeneous and therefore you have to poke at bits and pieces of some of these problems and then hope that at some point one can synthesize. Let me give you an illustration of something I wouldn't want to happen again. One of the rationales for the development of ADA was that there were on the order of two thousand different languages being used in the Department of Defense. Now, I don't know where that number came from, but by my standards it's sheer nonsense. I think what was happening was that every different assembly language was being counted, every compiler was being counted, and every time anybody twiddled a bit anywhere was being counted as one of the numbers that went into this two thousand count. I submit that if we knew what a language was, if we knew how to measure how many of them we had, we would be a lot better off. I would also suspect that when large industrial organizations, regardless of whether their applications were business oriented or scientifically oriented, go to their management and say they are using three languages, or thirty-three, it would be better if they could all agree on what it meant when they counted to three or to thirty- three. That's again a cost factor.

**Marvin Denicoff, Office of Naval Research**

Certainly the last two questions were really the same question, and underscored the obvious, namely; we need an a priori specification of purpose, of objectives, of profound versus trivial issues, et cetera. It's not enough, obviously, to simply collect data. I made a note, because, unless we know what data to collect, what level of precision, from whom we're collecting the data, for whom we're collecting the data, for what purpose, for how long, for what decision models, unless all of those things are specified a priori, we tend to get into trouble — and if you've been involved in data collection programs for areas other than software, you know how much trouble you can get into. However, a word of caution, and that's really the purpose of my comment; namely, if we knew a hell of a lot about all of these elements I've talked about, if we could pre-specify all of these parameters, we'd be in great shape. However, my argument is that we know nothing, almost nothing, and that there is a hell of a lot of insights that come out of *overcollecting*. Looking at the data per se sometimes yields profound insights into how we do this a priori specification out there in the future. So I say at the beginning, in the inception period of this data collection program, I would argue that to gain the advantage of those surprises we ought to err in the direction of overcollection.

**Alan Perlis**

A question was asked before about why we are collecting this data about languages if documentation is such an important part of our work. I wonder why nobody has gotten up and said, "Isn't there something wrong with computing if saying what a line of code does is so much more extensive an effort than the writing of the code itself?" For whom are we writing the documentation? And why are we writing it? It's actually an insurance policy that we use to reduce the cost of catastrophe. The assumption is always made that a large number of the people involved in software are in management and not in code production. And one of the reasons for that too, possibly, is the language tools we use. That is, it's not obvious at all that the amount of documentation in a piece of code is independent of the way we program and the languages we use.

**Capers Jones, ITT**

I'm assistant director of programming measurements for ITT, and my job is to measure program quality, productivity, and other attributes for all programs put out by the corporation, some thousands of them. I've been in measurement for a long time. Let me give you a quick dump of how you ought to go about it.

First, overcollection of data is very common when people get set to measure software. The way you avoid that, or at least minimize it, is to start by trying to come up with the format of the output reports to be presented to the managers or the programmers, the people you want to convince that you've got valid information. Then you work backwards from the output format to the data base and the input requirements. If you do it the other way, if you set out trying to specify the input of the data base fist, you almost always overcollect, and you also have another problem. You occasionally leave out important variables that you should collect. IBM's federal systems division did that. They set up a data collection effort that was so massive that after the first wave of inputting data individual managers stopped doing it because it was too time-consuming and too bothersome. You've got to have some simplification. But the purpose of all this is to make things better, not just to record data and do academic research forever. Generally speaking, what you want to make better are costs, quality, and schedules. So you aim your data collection system to feed back the information on those three significant variables. And the only thing that seems constant across all sizes and kinds of programs in this very diverse industry is the fact that defect removal costs more than anything else. One of the reasons why language selection is important, even though the direct cost of coding is small, like three percent on a big system, is that the choice of language can minimize defect removal later and defect removal is a very large cost, from thirty to thirty-five to forty percent of development.

Second, the paperwork costs are extremely significant for medium to large systems. They're hardly significant at all for small programs. Another point about paperwork is that the range in the paperwork domain is astonishing, and seems to be due to individual human variances. For example, the smallest logic spec I ever saw was two pages per thousand lines of code and the biggest — and it was assembler that was the target language — the biggest was sixty-eight, which was a variance of thirty-four to one with no relationship to complexity or the language. And paperwork is a controllable variable, and most companies do it too much. As Doctor Perlis suggested, it's to convince management that we really know what we're doing rather than because we really need it. So in the big system, while cutting down on paperwork actually is cost-effective and doesn't diminish the coding speed or the coding efficiency, it is a variable that gets out of control.

Measurement isn't a very tricky, difficult intellectual area, but if you try and think practically, then you only want to measure things that you can improve.