Technical Report TR-1290 May 1983

THE SOFTWARE INDUSTRY: A STATE OF THE ART SURVEY

Marvin V. Zelkowitz
Raymond Yeh
Richard G. Hamlet
John D. Gannon
Victor R. Basili

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

1. Introduction

1.1. Goals

The term "software engineering" first appeared in the late 1960s [Naur and Randell 69], [Buxton and Randell 70] to describe ways to develop, manage, and maintain software so that the resulting products are reliable, correct, efficient, and flexible. After 15 years of study by the computer science community, it is important to assess the impact that numerous software engineering advances have had on actual software production. The IBM Corporation asked the University of Maryland to conduct a survey of different program development environments in industry in order to determine the state of the art in software development and to ascertain which software engineering techniques are most effective in the non-academic sector. This report contains the results of that survey.

1.2. The Survey Process

This project began during the spring of 1981. The goal was to sample 15 to 20 organizations, including the primary sponsor of this project - IBM, and study their development practices. This was accomplished via a two-step process. A detailed survey form was sent to each of the participating companies. In response to the return of this form, a follow-up visit was made. This visit clarified the answers given on the form. We believe that this process, although limiting the number of places surveyed, resulted in more accurate information being presented than if we had just relied on forms.

Each survey form contains two parts. Section one asks for general comments concerning software development for the organization as a whole. The information described by this part typically represents the "standards and practices" document for the organization. In addition, we also studied several recently completed projects within each company. Each such project completed the second section of the survey form, which described the tools and techniques that were used on that particular project.

A variety of organizations in both the United States and Japan participated in the study. The acknowledgement at the end of this report lists some of the participants. Due to the proprietary nature of part of the information we obtained, some of the participants wish to remain anonymous. Over the life of this project, we surveyed 25 different organizations. Thirteen of them are U.S. companies and 12 were from Japan. Due to the cost and time restrictions, about half of the Japanese companies were not interviewed, and the other half were interviewed in varying degrees of detail.

In addition to our survey form, interviews were held with several company officials, and some published references were used for additional data. Figure 1 lists the basic data processed. In order to characterize the projects we studied, projects and teams are somewhat arbitrarily classified into four groups according to sizes: Small, Medium, Large, and Very Large.

Projects are classified according to the number of staff months needed to complete them, and teams according to the numbers of members. This division leads to a breakdown in which there is only one case of a team that is larger than a project (Company U).

Code	Divs	Projs	Interview	Project Size	Team Size
A	2	3	Yes	L	L
B	1	0	No	S	VL
C	1	1	No	S	M
D	1	3	Yes	L	L
E	3	4	Yes	VL	VL
F	1	3	Yes	VL	VL
G	1	2	Yes	L	L
H	1	7	Yes	L	M
I	1	9	Yes	VL	VL
J	1	4	Yes	L	VL
K	1	8	Yes	VL	M
L	1	1	Yes	L	VL
M	1	3	Yes	M	VL
N	1	2	No	S	S
O	1	1	Yes	VL	VL
P	1	1	No	M	-
Q	2	0	No	M	L
R	1	0	No	-	-
S	1	1	Yes	M	S
T	1	4	Yes	VL	VL
U	1	0	Yes	L	VL
V	1	1	Yes	M	S
W	1	1	Yes	L	S
X	1	1	No	L	S
Y	1	1	No	L	-

Figure 1.a Companies Surveyed

Project Size (staff months)	Team Size (staff)
S<10	S<10
M 10-100	M 10-25
L 100-1000	L 25-50
VL>1000	VL>50

Figure 1.b Legend

After reviewing the basic data, we recognized three different software development environments:

(1) Contract software - Typically Department of Defense and NASA aerospace systems

(2) Data processing applications - Typically software produced by an organization for its own internal business use

(3) Systems software - Typically operating system support software produced by a hardware vendor as part of a total hardware-software package of products for a given operating system.

A single company might be represented in more than one of the above categories. For example, we looked at several Defense-related projects and one internal data processing application at an aerospace company.

This survey is not meant to be all-encompassing; however, we believe that we have surveyed a large enough number of locations to understand software development in industry today. Several companies were concerned about which projects we should study -- we left that decision up to them. There was concern that the projects we were looking at were "not typical" of the company. (Interestingly, very few companies claimed to be doing "typical" software.) We felt that we were getting to see the "better" developed projects. In general, every company had either a written guideline or unwritten folklore as to how software was developed. Deviations from this policy were rare.

2. General Observations

The literature contains many references to software engineering methodology, including tools support throughout the lifecycle, language support in other than source code, testing support, measurement and management practices, and other techniques that will be mentioned throughout this report. But in our survey, we found surprisingly little use of software engineering practices across all companies. No organization fully tries to use the available technology. While some companies had stronger management practices than others, none used tools to support these practices in any significant way.

2.1. Organizational Structure

Most companies that we surveyed had an organizational structure similar to the one in Figure 2.

```
                              Director of Software
                                       !
   ------------------------------------------------------------------------
   !                !                  !          . . .                  !
Software           Area               Area                              Area
Technology          1                  2                                 N
Group            Managers           Managers                          Managers
```

Figure 2 Typical Organization Structure

The software technology group typically has one to five individuals collecting data, modelling resource usage, and generating standards and practices documents. However, this group has no direct authority to enforce adherence to software engineering practices even within a single division. As a result, standards often vary within a single organization.

This structure also explains a current anomaly in the use of software engineering techniques. Although they are frequently mentioned in the literature and at conferences, software engineering techniques are rarely used correctly by the industry at large. Developers of real products often think that the software technology (research) group of Figure 2 (who are the conference attendees and write most of the research papers) is too optimistic about the effects of these techniques and are unrealistic since they have not applied them to real life situations. Managers know their personnel often lack the education and experience needed for successful applications of these techniques. Even the techniques that have been adopted are frequently misused. For example, although many companies used the term "chief programmer" to describe their programming team organizations, most bore little resemblance to the technique described in the literature. Generally each project had two to three levels of management who handled staff and resource acquisition, but who did not actively participate in system design.

A further problem in many organizations is that there is generally no one person at the head of the chart of Figure 2 who makes software decisions. Such a person often exists in hardware organizations. For this reason, software standards are generally low and vary across the company.

2.2. Tool Use

Tool use is relatively low across the industry. Not too surprisingly, the use of tools varies inversely from their "distance" from the code and unit test phase of development. That is, tools are most frequently used during the code and unit test phase of software development (e.g., compilers, code auditors, test coverage monitors, etc.). Adjacent phases of the software lifecyle, design and integration, usually have less tool support (e.g., PDL processors and source code control systems). Few requirements or maintenance tools are used. In looking at tool use, Figure 3 gives some indication of which techniques and tools are used:

Method or Tool	Per Cent of Companies
High-level languages	100*
Online access	89
Reviews	84
Program design languages	60
Formal methodology	45
Test tools	27
Code auditors	18
Chief programmer team	13
Formal verification	0
Formal requirements or specifications	0

*-Every company seems to use some higher level language-but often there is also a high usage of assembly language.

Figure 3 Industrial Method or Tool Use

Time sharing computer systems and compiler writing became practical in the late 1960s and early 1970s; thus online access and high level languages can probably be labeled the successes of the 1960s. Similarly, the widespread use of reviews and pseudocode or program design language (PDL) permits us to call them the successes of the 1970s. It is disappointing that few other tools have been adopted by industry. Testing tools are used by only 27% of the companies, and most of these are simply test data generators. Only one company (in Japan) indicated that it used any form of unit test tool to measure test case coverage. Although many companies claim to use chief programmer teams, few actually do.

While PDLs are heavily used, it is disappointing that the process is not automated. Some PDL processors are simply manual formatters, while some do "pretty print" and indent the code. Often the PDL is only a "coding standard" and not enforced by any tool. Only one location had a PDL processor that checked interfaces and variable use/define patterns.

Tool use generally has the flavor of vintage 1970 time sharing. Jobs have a "batch flavor" in that runs are assembled and then compiled. There is little interactive computing. There is minimal tool support - mostly compilers and simple editors.

> The problems in using tools can be attributed to several factors. Corporate management has little (if any) software background and is not sympathetic with the need for tools. No separate corporate entity exists whose charter includes tools so there is no focal point for tool selection, deployment and evaluation. Tools must be paid out of project

funds, so there is a fair amount of risk and expenditure for a project manager to adopt a new tool and train his people to use it. Since project management is evaluated on meeting current costs and schedules, and tool use must be amortized across several projects to be effective, a single project manager will almost always stand out as "unproductive." Companies often work on different hardware, so tools are not transportable, limiting their scope and their perceived advantage. The most striking example of this, was one system where $1M was spent building a data base, yet there was no thought of ever using that data base on another system. The need to maintain large existing source code libraries (generally in assembly code) makes it hard to introduce a new tool that processes a new higher level language. Finally, many of the tools are incomplete and poorly documented. Because such tools fail to live up to promises, project managers are justifiably reluctant to adopt them or consider subsequently developed tools.

2.3. Japan - U. S. Comparisons

There is currently much interest in comparing U. S. and Japanese technology. In general, development practices are similar. Programmers in both countries complain about the amount of money going towards hardware development and the lack of resources for software. However, in comparing U.S. and Japanese software development, we found that Japanese companies typically optimize development across the company rather than within a single project. One effect of this is that tools become a capitalized investment paid for or developed out of company overhead rather than project funds. The cost of using tools is spread among more projects, knowledge about tools is known to more in the company, and project management is more willing to use tools since the risk is lower. Thus, tool development and use is more widespread in Japan.

2.4. Reviews

At the end of each phase (and sometimes within a phase) the evolving software product (i.e., requirements, design, code, test cases, see for example [Belady and Lehman 76]) is subjected to a review process, trying to uncover problems as soon as possible. ("Inspection" and "walkthrough" [Fagan 76] are other terms used for reviews without regard to the distinctions made in the software-engineering literature.) Nearly everyone agrees that reviews work, and nearly everyone uses them, but there is a wide variety in the ways that reviews are conducted. There seems to be an agreement that they allow the routine completion of software projects within time and budget constraints that only a few years ago could be managed only by luck and sweat. Reviews were first instituted for code, then extended to design. Extensions to requirements and test-case design are not universal, and some feel that the technique may have been pushed beyond its usefulness. Managers would like to extend the review process, while the technical people are more inclined to limit it to the best-understood phases of development.

Two aspects of reviews must be separated: one is management control and the other is technical utility. Managers must be concerned with both aspects, but technical success cannot be assured by insisting that certain forms be completed. If the tasks assigned to the reviewers are ill-defined, or the form of the product reviewed inappropriate, the review will waste valuable people's time. Lower-level managers prefer to use reviews where they think reviews are appropriate, and avoid them in other situations.

The technical success of the review process rests squarely on the expertise and interest of the people conducting the review, not on the mechanism itself. The review process is refined by continually changing it to reflect past successes and failures, and much of this information is subjective, implicitly known to experienced participants. Some historical information is encoded in review checklists, which newcomers can be trained to use. However, subjective items like the "completeness" of requirements are of little help to a novice.

2.5. Data Collection

Every company collects some data, but little data becomes part of the corporate memory to be used beyond the project on which it was collected. Data generally belongs to individual managers, and it is their option as to what to do with it. Data is rarely evaluated and used in a postmortem analysis of a project. After a project is completed, it is rarely subjected to an analysis to see if the process could have been improved. This is not the case in Japanese companies, in which postmortem analysis was more frequently performed.

Several companies are experimenting with various resource models (e.g., SLIM [Putnam 79], PRICE S [Freiman and Park 79], etc.). No company seems to trust any model enough to use it on a full proposal; instead the models are used to check manual estimates. Figure 4 shows that little data is being collected across all companies.

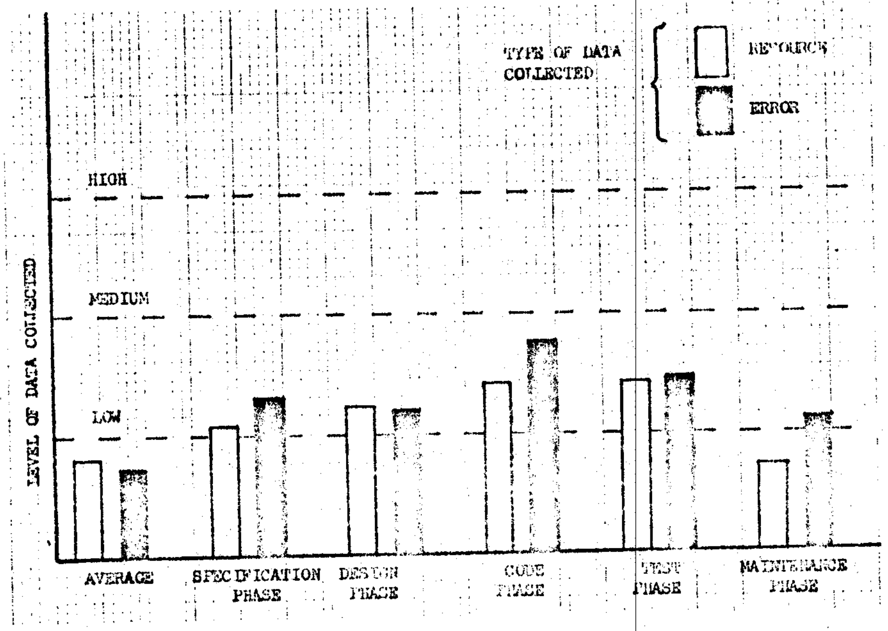

Figure 4 Data Collection Across Life Cycle

In general, we found it extremely difficult to acquire data. First of all, quantitative data is quite rare within most companies. In addition each company has different definitions for most of the measured quantities, such as:

(1) Lines of code is defined as source lines, comment lines, with or without data declarations, executable lines or generated lines.

(2) Milestone dates depend on the local software life cycle used by the company. Whether requirements, specification, or maintenance data is included will have a significant effect upon the results.

(3) Personnel might include programmers, analysts, typists, librarians, managers, etc.

Much of this data is proprietary. The differing definitions of quantities for which data was collected prevent any meaningful comparison. It is quite evident that the computer industry needs more work on the standardization of terms in order to be able to address these quantitative issues in the future.

3. Software Development Environments

In the following section general characteristics about most software environments are described. The last sections outline particular characteristics of the three classes of environments that we studied in detail.

3.1. General Life Cycle

3.1.1. Requirements and Specification

In all places we contacted, requirements were in natural language text (English in the US and Kanji in Japan). Some projects had machine-processable requirements documents, but tool support was limited to interactive text editors. No analysis tools (e.g., SREM [Alford 77], PSL/PSA [Teichroew and Hersey 77]) were used except on "toy" projects. Projects were either too small to make the use of such a processor valuable, or else too large to make use of the processor economical.

Reviews determine if the system architechure is complete, if the specifications are "complete", the internal and external interfaces are defined, and the system can be implemented. These reviews are the most difficult to perform and their results are highly dependent on the quality of people doing the review because the specifications are not formal. There is little traceability between specifications and designs.

3.1.2. Design Phase

Most designs were expressed in some form of Program Design Language (PDL) or pseudocode, which made design reviews effective. Tools that manipulated PDL varied from editors to simple text formatters. Only one company extended its PDL processor to analyze interfaces and the dataflow in the resulting design.

While using PDL seems to be accepted practice, its effective use is not a foregone conclusion. For example, we consider the expansion of PDL to code a reasonable measure of the level of a design. A 1:1 PDL to source code expansion ratio indicates that the design was essentially code instead of design. Figure 5 indicates the ranges of expansions of PDL to code found at several locations that provided such data.

Location	PDL to Code Ratio
1	1 to 5-10
2	1 to 3-10
3	1 to 1.5(!!)-3

Figure 5 Expansion of PDL to Code

Customer involvement with design varied greatly even within installations. Producing lots of detailed PDL is much the same as producing lots of detailed flowcharts. (Nobody cares, but it's in the contract.)

3.1.3. Code and Unit Test

Most code that we saw was in higher level languages -- Fortran for scientific applications or some local variation of PL/I for systems work.

In the aerospace industry FORTRAN was the predominant language. People who normally worked in assembly language thought that FORTRAN and PL/I significantly enhanced their productivity. Historical studies have shown that programmers produce an average of one line of debugged source code per hour regardless of the language. ([Brooks 75] contains a concise review of this work.)

Despite claims that they used chief programmer teams in development, very few first or second-line managers ever wrote any PDL, or code themselves. We heard complaints that chief programmer teams worked well only with small groups of 6-9 people, and on projects in which a person's responsibility was not divided between different groups.

Much of the code and unit test phase lacks proper machine support. Code auditors could greatly enhance the code review process. We studied one code review form and found that 13 of 32 checks could be automated. Manual checks are currently performed for proper indentation of the source code, initialization of variables, interface consistency between the calling and called modules, etc.

Most unit testing could be called adversary testing. The programmer claims to have tested a module and the manager either believes the programmer or not. No unit test tools are used to measure how effectively the tests devised by a programmer exercise his source code. While a test coverage measure like statement or branch coverage is nominally required during the review of unit test, mechanisms are rarely available to assure that such criteria have been met.

3.1.4. Integration Test

Integration testing is mostly stress testing -- running the product on as much real or simulated data as is possible. The data processing environment had the highest level of stress testing during integration testing. Systems software projects were relatively slack in integration testing compared to the banking industry.

3.2. Resources

Office space for programmers varied from 1 to 2 programmers sharing a "Santa-Teresa" style office [McCue 1978] with a terminal to large bullpens divided by low, moveable partitions. Terminals were the dominant mode for computer access. Some sites had terminals in offices, while others had large terminal rooms. The current average seems to be about two to seven programmers per terminal. Within the last two years most companies have realized the cost-effectiveness of giving programmers adequate computer access via terminals, but have still not provided adequate response time. Ten to twenty second response time was considered "good" at some places, where sub-second response could be used [Thadani 82].

> It seems worth noting that most companies were willing to invest in hardware (e.g., terminals) to assist their programmers, but were reluctant to invest in software that might be as beneficial.

3.3. Education

Most companies had agreements with a local university to send employees for advanced training (e.g., MS degrees). Most brought in special speakers. However, there was little training for project management. Only one company had a fairly extensive training policy for all software personnel.

Many companies had the following problems with their educational program:

(1) Programmers were sent to courses with little or no follow-up experience. Thus what they learned was rarely put into practice, and was often forgotten.

(2) Some locations complained about their distance from any quality university, and the difficulties that such isolation brought.

3.4. Data Collection Efforts

The data typically collected on projects includes the number of lines of PDL for each level of design, the number of lines of source code produced per staff-month, the number and kinds of errors found in reviews, and a variety of measures on program trouble reports. The deficiency of lines of code as a measure can be indicated by the range in values of "good" developments, as given by Figure 6:

LOC/Staff-Month	Application and Language
75	OS in Assembler
91	I/O controller in HLL
142-167	OS in HLL
182-280	Assembler applications

Figure 6 Source Code per Staff-Month

Due to the differing application areas, it is not really possible to compare these numbers. However, it does seem obvious that the difficulty of the application area (e.g., operating systems and other real-time programs being the most difficult) has more impact on productivity than does the implementation language used.

One location reports the following figures for errors found during reviews.

Phase	Defects/1000 Lines
Design	2 major, 5 minor
Code	5 major, 8 minor

Figure 7 Defects Discovered During Reviews

The classification of errors into categories like "major" and "minor" is actuarial. While the classification is useful for putting priorities on changes, it sheds little light onto the causes and possible treatments of these errors.

4. Three Development Environments

4.1. Applications Software

We studied 13 projects in 4 companies that produce applications software. In this area, software is contracted from the organization by a Federal agency, typically the Department of Defense or NASA. Software is developed and "thrown over the wall" to the agency for operation and maintenance. Typically, none of the organizations we surveyed were interested in maintenance activities. All believed that the payoff in maintenance was too low, and smaller software houses could fill that void.

Since contracts are awarded after a competitive bidding cycle (after a Request For Proposal) and requirements analysis is typically charged against company overhead, analysis was kept to a minimum before the contract was awarded. In addition, since the goal was to win a contract, there was a clear distinction between cost and price. _Cost_ was the amount needed to build a product - a technical process at which most companies believed they were reasonably proficient. On the other hand, _price_ was a marketing strategy needed to win a bid. The price had to be low enough to win, but not too low to either lose money on the project or else be deemed "not responsive" to the requirements of the RFP. Thus many of the ideas of software engineering developed during the 1970s on resource estimation and workload characterization are not meaningful in this environment due to the competitive process of winning bids.

In addition, two distinct types of companies emerged within this group - _system developers_ and _software developers_. The system developers would package both hardware and software for a government agency, e.g., a communications network. In this case, most of the costs were for hardware with software not considered significant. On the other hand, the software developers simply built systems on existing hardware systems. DEC's PDP/11 series seemed to be the most popular with system builders that were not hardware vendors.

All of the companies surveyed had a methodology manual; however, they were either out of date, or were just in the process of being updated. In this environment, Department of Defense MIL specifications were a dominant driving force and most standards were oriented around government policies. The methodology manuals were often policy documents outlining the type of information to be produced by a project, but not how to obtain that information.

As stated previously, most organizations bid on RFPs from government agencies. Because of that, requirement analysis is kept to a minimum. Requirements are written in English and no formal tool is used to process the requirements.

Except for one company, FORTRAN seemed to be the dominant programming language. Two tools did seem to be used. Due to DoD specifications, most had some sort of management reporting forms on resource utilization. However, these generally did not report on programmer activities. PDL was the one tool many companies did depend on - probably because the cost was low.

Staff turnover was uniformly low - generally 5% to 10% a year. Space for programmers seemed adequate, with 1 to 2 per office being typical. All locations, except one, used terminals for all computer access, and that one site had a pilot project to build "Santa Teresa"-style offices connected to a local minicomputer.

4.2. Systems Software

We studied eight projects produced by three vendors. All of the projects were for large machines, and operating systems for those machines were the most important projects studied. The other projects, mostly compilers and utilities, did not follow the same development rules as did operating systems projects, because they were considered to be small and their designs well-understood.

The software is generally written on hardware similar to the target machine. Terminals are universally used and the ratio of programmers to terminals varies from almost 1:1 to 3:1. Getting a terminal is frequently less of a problem than getting CPU cycles to do development.

Software support is generally limited to line-oriented text editors and interactive compilers. High-level development languages exist, and in most cases there is a policy that they be used; however, a substantial portion of operating systems remains in assembler language (20% to 90% depending upon company). The reasons are partly good ones (such as the prior existence of assembler code) and partly the usual one: alternatives have never been considered at the technical level. Text formatting programs are in wide use, but analysis of machine-readable text other than source code is virtually nonexistent.

Most testing is considered part of the development effort. There may be a separate test group, but it reports to the development managers. Only a final "field" test may be under the control of an independent quality-control group.

Maintenance is usually handled by the development staff. A field support group obtains trouble reports from the field, and then forwards them on to the development organization for correction. In many cases, the developers, even if working on a new project, handle errors.

Programmers are usually organized into (usually) small teams by project, and usually stick with a project until it is completed. The term "chief programmer team" is used incorrectly to describe conventional organizations: a chain of managers (the number depends on project size) who do not program, and small groups of programmers with little responsibility for organization.

Staff turnover is relatively high (up to 20% per year) compared to the applications software area. Most programmers typically have private cubicles parcelled out of large open areas. The lack of privacy is often stated as a negative factor.

Software engineering practices vary widely among the projects we investigated. There was a strong negative correlation between the age of the system and the amount of software engineering used.

4.3. Data Processing

We studied 6 data processing projects at 4 locations, although every location had some data processing activities for its own internal use. Most data processing software that we studied was developed in COBOL, although some systems are written in FORTRAN, and used to provide internal data processing services for the company. These systems did not produce revenue for the company, and were all "company overhead." There was a need to maintain the code throughout the life cycle.

Requirements were mostly in English and unstructured, although one financial company spructured specifications by user function. Designs, especially for terminal-oriented products, were relatively similar - a set of simulated screen displays and menus to which the user could respond. The most striking difference in the data processing environment was the heavy involvement of users in the two development steps. The success of the project depended upon the degree of user involvement <u>before</u> integration testing. One site clearly had a "success" and a "failure" on two different projects that used the same methodology. The company directly attributed the success and failure to the interest (or lack of interest, respectively) to the user assigned to the development team during development.

All usage that we observed was via terminals. Office space was more varied than in the other two environments we observed. Some places used one and two-person offices, while others partitioned large open areas into cubicles. "Stress" was often high in that overtime was more common, and turnover was the highest in this environment - often up to 30% per year, although one location had a low turnover rate which they attributed to relatively higher salaries than comparable companies.

Data processing environments often used a phased approach to development, and quality control was especially important. One location, which had numerous failures in the past, attributed their recent successes to never attempting any development that would require more than 18 months. Since these systems often managed the company's finances, the need for reliability was most critical and stress testing was higher than in other areas.

5. Conclusions

We feel there are both short and long-term remedies to raise the level of methodology and tool use throughout industry. The short-term suggestions are relatively conservative; however, we feel they can improve productivity. While we can point to no empirical evidence that will permit us to forecast gains, there is a general consensus in the software community (like that for the use of high level languages) to support these ideas. Our long-term suggestions could form the basis for a research effort.

5.1. Short Term

(1) More and better computer resources should be made available for development. The computer systems being used for

development are comparable with the best of those available in the late 1960's or early 1970's: timesharing on large machines. The use of screen editors at some locations has been a major improvement, but other tools seem limited to batch compilers and primitive debugging systems. Response time seems to be a major complaint at many development installations.

(2) Methods and tools should be evaluated. A separate organization with this charter should be established. As of now, it does not appear that any one group in most companies has the responsibility to study the research literature and try promising techniques. Since the most successful tools have been high level language compilers, the first tools to be developed should be integrated into compilers. Thus these tools should concentrate on the design and unit test phases of development during which formal languages exist and compiler extensions are relatively straightforward. This organization could both acquire and evaluate the tools via case studies and/or experiments.

(3) Tool support should be built for a common high level language. The tools we would pick first include a PDL processor, a code auditor, and a unit test coverage monitor. The PDL processor should at least check interfaces. Unfortunately, commercially available processors do little more than format a listing; however, interface checking is nothing more than 20-year-old compiler technology. The

processor should also construct graphs of the flow of data through the design and extract PDL from source code so that while both are maintained together they can be viewed separately. Code auditors can be used to check that source code meets accepted standards and practices. Many of these checks are boring to perform manually (e.g., checking whether BEGIN-END blocks are aligned) and thus become error prone. Unit testing tools can evaluate how thoroughly a program has been exercised. These tools are easy to build and should meet with quick acceptance since many managers require statement or branch coverage during unit test.

PDL processors should support an automated set of metrics that cover the design and coding process. The metrics in turn can monitor progress, characterize the intermediate products (e.g., the design, source code, etc), and attempt to predict the characteristics of the next phase of development. Possible metrics include design change counts, control and data complexity metrics for source code, structural coverage metrics for test data, etc. [Basili 80].

(4) <u>Improve the review process</u>. Reviews or inspections are a strong part of current methodology. The review process can be strengthened by the use of the tools mentioned above. This would permit reviewers to spend more time on the major purpose of the review -- the detection of logical errors, and avoid the distractions of formatting or syntactic anomalies.

(5) Use incremental development (e.g., iterative enhancement [Basili and Turner 75]). One data processing location, after repeatedly failing to deliver software, made a decision never to build anything that had a chunk larger than those requiring 18 staff months. Since then they have been successful.

(6) Collect and analyze data. Most of the data being collected now is used primarily to schedule work assignments. Measurement data can be used to classify projects, evaluate methods and tools, and provide feedback to project managers. Data should be collected across projects to evaluate and help predict the productivity and quality of software. The kind of data collected and analysis performed should be driven by a set of questions that need answers rather than what is convenient to collect and analyze. For example, classifying errors into "major" and "minor" categories does not answer any useful questions. A more detailed examination of error data can determine the causes of common errors, many of which may have remedies. Project post mortems should be conducted.

5.2 Long Term

(1) Compiler technology should be maintained. Many companies seem to "contract" out compiler development to smaller software houses due to "pedestrian" nature of building most compilers. While compiler technology is relatively

straightforward and perhaps cheaper to contract to a software house, the implications are far reaching. Software research is heading towards an integrated environment covering the entire life cycle of software development. Research papers are now being written about requirements and specification languages, design languages, program complexity measurement, knowledge based Japanese "fifth generation" [Karatsu 82] languages, etc. All of these depend upon mundane compiler technology as their base.

(2) Prototyping should be tried. It was never mentioned during our visits.

(3) Develop a test and evaluation methodology. Test data has to be designed and evaluated. While the current software development process provides for the design of test data in conjunction with the design of the software, there is little tool support for this effort. As a result, almost every project builds its own test data generator and a few even build test evaluators. Concepts like attribute grammars may provide the basis for a tool to support test data generation.

(4) Examine the maintenance process. The maintenance process should be formalized as part of the continuing development process. Maintenance was rarely mentioned in our interview process, although there is a project in Japan to build maintenance workstations. Their view is that development is

a subset of maintenance. This implies that the successful methods and tools used in development should be adapted for use in this stage of the process.

(5) Encourage innovation. Experimental software development facilities are needed. Management should be encouraged to use new techniques on small funded-risk projects.

6. Acknowledgements

This project was sponsored by a contract from the IBM Corporation to the University of Maryland. We also wish to acknowledge the cooperation of the following organizations in addition to IBM for allowing us to survey their development activities: Bankers Trust Company, Honeywell Large Information Systems Division, Kozo Keikaku Kenkyujo, Japan Information Processing Service, Nomura Computer Systems Ltd., Software Research Associates (Japan), Sperry Univac, System Development Corporation, Tokyo Electric Power Company, Toshiba Corporation, TRW, Xerox, and several other organizations who wish to remain anonymous. This project would not have been possible without their help.

7. References

[Alford 77] M. W. Alford. A Requirements Engineering Methodology for Real-time Processing Requirements. IEEE Transactions on Software Engineering SE-3, 1, (January, 1977), 60-69.

[Basili 80] V. R. Basili. Models and Metrics for Software Management and Engineering. IEEE Computer Society Press, 1980.

[Basili and Turner 75]
V.R. Basili and A.J. Turner. Iterative Enhancement: A Practical Technique for Software Development. IEEE Transactions on Software Engineering SE-1, 4, (December 1975), 390-396.

[Belady and Lehman 76] L. A. Belady and M. M. Lehman. A model of Large Program Development. IBM Systems Journal 15, 3, (September, 1976), 225-252.

[Boehm 81]
B.W. Boehm. Software Engineering Economics. Prentice-Hall, Inc., Englewood Cliffs, N.J., (1981).

[Brooks 75]
F.P. Brooks, Jr. The Mythical Man-Month. Addison-Wesley Publishing Company, Reading, MA, (1975).

[Buxton and Randell 70]
J.N. Buxton and B. Randell (ed.). Software Engineering Techniques. NATO Scientific Affairs Division, Brussels, (1970).

[Fagan 76]
M.E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal 15, 3, (1976), 182-211.

[Freiman and Park 79]
F. Freiman and Park. PRICE Software Model Overview. RCA, (February 1979).

[Halstead 77]
M.H. Halstead. Elements of Software Science. Elsevier, New York, (1977).

[Karatsu 82]
H. Karatsu, What is required of the 5th generation computer - social needs and its impact, Fifth Generation Computer Systems, North Holland, 1982.

[McCue 1978]
G. M. McCue, IBM's Santa Teresa Laboratory - Architectural design for program development, IBM Systems Journal 17, 1 (1978) 4-25.

[Naur and Randell 69]
P. Naur and B. Randell (ed.). Software Engineering. NATO Scientific Affairs Division, Brussels, (1969).

[Putnam 79]
L. Putnam. SLIM Software Life Cycle Management Estimating Model: User's Guide. Quantitative Software Management, (July 1979).

[Teichroew and Hersey 77]
D. Teichroew and E.A. Hersey III. PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems. IEEE Transactions on Software Engineering, SE-3, 1, (January 1977), 41-48.

[Thadani 81]
A.J. Thadani. Interactive User Productivity. IBM Systems

Journal, **20**, 4, (1981), 407-423.