This in-depth survey of 30 companies reveals actual goings-on in software production. Results show that, while practice is 10 years behind research, we have the tools to narrow the gap.

Software Engineering Practices in the US and Japan

Marvin V. Zelkowitz, Raymond T. Yeh, Richard G. Hamlet, John D. Gannon, and Victor R. Basili,
University of Maryland

The term *software engineering* first appeared in the late 1960's to describe ways to develop, manage, and maintain software so that resulting products are reliable, correct, efficient, and flexible.[1] The 15 years of software engineering study by the computer science community has created a need to assess the impact that numerous advances have had on actual software production. To address this need, IBM asked the University of Maryland to conduct a survey of different program development environments in industry to determine the state of the art in software development and to ascertain which software engineering techniques are most effective. Unlike other surveys, such as the recent one on Japanese technology,[2] we were less interested in recent research topics. Journals, such as the *IEEE Transactions on Software Engineering* adequately report such developments; we were more interested in discovering which methods and tools are actually being used by industry today.[3] This report contains the results of that survey.

The goal of this project, which began in spring 1981 and continued through summer 1983, was to sample about 20 organizations, including IBM, and study their development practices. We contacted major hardware vendors in the US, and most agreed to participate. Several other software companies and other "high-technology" companies were contacted and agreed to participate. While we acknowledge that this survey was not all inclusive, we did study each company in depth, and based on discussions with others in the field, we believe that what we found was typical.

We were not interested in R&D activities in these companies. Most had individuals engaged in interesting developments, and most knew what was current in the field. Our primary concern was what the average programmers in these companies did to develop software projects.

Data was collected in a two-step process. A detailed survey form was sent to each participating company. When the form was returned, a follow-up visit was made to clarify the answers given. We believe that this process, although limiting the number of places surveyed, allowed us to present more accurate information than if we had relied on the returned forms alone.

Each survey form contained two parts. Section one asked for general comments on software development for the organization as a whole. The information typically represented the *standards and practices* document for the organization. In addition, several recently completed projects within each company were studied. Each project leader completed the second section of the survey form, which described the tools and techniques used on that project.

Several companies were concerned that the projects we were looking at were not typical of them. (Interestingly, very few companies claimed to be doing typical software.) However, since the companies selected the projects they described on the form, we believe we saw the better developed projects—if there is any bias to our report, it is that the average industry project is probably worse than what we describe here.

Thirty organizations in both the US and Japan participated in the study: five IBM divisions, 12 other US companies, and 13 Japanese companies. About half the Japanese companies were not interviewed, while the other half were interviewed to varying degrees of detail. All US companies were interviewed. The "Acknowledgments" section at the end of this article lists the US participants. Some of the Japanese participants never responded to our request for permission to use their names, so only a few Japanese companies are listed.

Table 1 characterizes the companies visited, divisions within a company, and the projects studied, arbitrarily

0018-9162/84/0600-0057$01.00 © 1984 IEEE

classifying projects and teams 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 number of members. All companies listed with zero projects were Japanese companies that submitted part one of our form only. We interviewed at least one manager in depth in all surveyed US companies, in addition to general project management personnel.

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

(1) *contract software.* Department of Defense and NASA aerospace systems;
(2) *data processing applications.* Software produced by an organization for its own internal business use; and
(3) *systems software.* 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 have projects in more than one of these categories. For example, one aerospace company was involved in several DoD-related projects and one internal data processing application.

Table 1. Companies surveyed. The size of the project is in staff-months where (S)mall = < 10, (M)edium = 10-100, (L)arge = 100-1000, and (V)ery (L)arge = > 1000. Team size is in staff members where S = < 10, M = 10-25, L = 25-50, and VL = > 50.

CODE	NO. OF DIVISIONS	NO. OF PROJECTS	INTERVIEWED	PROJECT SIZE	TEAM SIZE
A	2	3	Yes	L	L
B	2	7	Yes	VL	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	-
Z	1	2	Yes	M	S
AA	2	5	Yes	VL	VL
BB	1	1	Yes	M	S
CC	1	1	Yes	L	S
DD	1	7	Yes	VL	VL

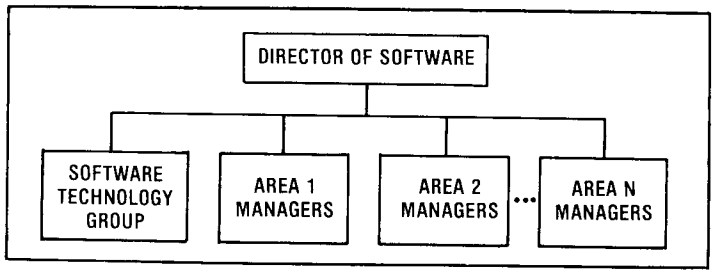


Figure 1. Typical organization structure.

General observations

This article is a series of general observations about each environment. Two of our first observations were that the data collected by each organization is insufficient and interpretations for similar concepts (e.g., phases of the life cycle, job descriptions from similar sounding titles, what certain automated tools should or did do, etc.) differ. We could have generated a survey consisting of 50 to 100 techniques and proceeded to tabulate them in a report. However, as we found out, the detailed interview process was much more informative. In addition, we would not want others quoting such numbers, since they would be subjective and imprecise. We believe that the structure we chose gives a better idea of software development today.

Every company had either written guidelines or unwritten folklore as to how software was developed, and major deviations were rare. Differences in projects within a company were less than the differences in projects among companies. But more significant was the wide gulf between practices in industry and those documented in current software engineering literature.

The literature contains many references to software engineering methodology, including tool support throughout the life cycle, specification and design languages, test data generators and coverage metrics, measurement and management practices, and other techniques. We found surprisingly little use of software engineering practices across all companies. No organization fully tries to use the available technology. Although some companies had stronger management practices than others, none used tools to support these practices in any significant way.

We are not implying that the companies do not have talented personnel. Most have individual projects that try to keep abreast of current technology, but within each company these projects are relatively rare, and the resulting experience is rarely applied to a different project.

Organizational structure. Most companies had an organizational structure similar to the one in Figure 1. The software technology group typically has one to five individuals collecting data, modeling resource usage, and generating standards and practices documents. However, this group has no direct authority to mandate software engineering practices even within a single division. As a result, standards often vary within a single organization.

We believe that this structure explains a current anomaly in the use of software engineering techniques. Developers of real products often think that members of the software technology (research) group (who attend na-

tional conferences and write most of the research papers) are too optimistic about the effects of these techniques, since they have not applied them to software products. Managers know their personnel often lack the education and experience needed to apply these techniques successfully. Even 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 descriptions bore little resemblance to the technique outlined in the literature. Many projects had two to three levels of managers who handled staff and resource acquisition but did not actively participate in system design.

A further problem in many organizations is that generally no one person is the Director of Software (see Figure 1), responsible for making software decisions. In some companies, software activities span several divisions. So even if such a person exists in each division, standards vary across the company. Interestingly, organizations often have one person making hardware decisions.

Tool use. Tools are not widely used in the industry. Not too surprisingly, the use of tools varies inversely from how far the tools are from the code development phase. Tools are most frequently used during the code and unit test phase of software development (e.g., compilers, code auditors, test coverage monitors, etc.). Tools are less frequently used in the adjacent phases of the software life cycle—design and integration (e.g., PDL processors and source code control systems). Few requirements or maintenance tools are used. Table 2 gives some indication of which techniques and tools are used. In classifying companies, we were somewhat subjective because we counted tools used on most projects, but not necesarily all, within a company. For example, every company used high-level languages on some projects, but they also used assembly language quite frequently.

Companies tend to adopt methods relatively early because their "capitalization" cost is relatively low, but tool use takes longer, since development or purchase costs are higher.

Although the percentages in Table 2 are not exact, the trends seem clear. Tool use generally has the flavor of vintage-1970 timesharing. Jobs have a batch flavor in that runs are built and then compiled. Tool support is minimal—mostly compilers and simple editors.

Timesharing computer systems and compiler writing became practical in the late 1960's and early 1970's; thus, on-line access and high-level languages can probably be labeled the successes of the 1960's. Similarly, since reviews and pseudocode or program design language (PDL) were so widely used, we can call them the successes of the 1970's. It is disappointing that few other tools have been adopted by industry. Testing tools are used by only 27 percent of the companies, and most are simply test-data generators. Only two companies used any form of unit test tool to measure test-case coverage. In a few cases, tools were developed but deemed too expensive to use. In one company, the quality assurance group had developed a testing tool and was trying to get it used on various projects. However, each project manager with whom we spoke praised the tool but claimed it was too expensive to run, increased the size of the system too much, etc.

PDL is frequently used, but it is not automated. Some PDL processors are simply manual formatters, while some do a 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 definition/use patterns for variables.

Our sample identified two general classes of companies. One class had strong management control over development with little tool support, while the other had relatively lax control over programmers, who generally built their own tools. In one extreme case, three project managers we interviewed expounded the virtues of their individual text editors.

The problems in using tools can be attributed to several factors. Corporate management, particularly of hardware vendors, tends to have an engineering background. Managers have little, if any, software background and are not sympathetic with the need for tools. No separate corporate entity exists whose charter includes tools, so we have no focal point for tool selection, deployment, and evaluation. Tools must be funded from project development budgets, so there is a fair amount of risk and expenditure for a project manager to adopt a new tool and train people to use it. Since project management is rated according to whether current costs and schedules are met, tool use must be amortized across several projects to be effective. Consequently, a project manager building and using a new tool 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 handicap was one system in which one million dollars was spent building a database, yet no one ever thought of using that database on another system. The need to maintain large existing products (written in the past in assembly code) makes it hard to introduce a new tool that processes a new higher level langauge. 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.

Review process. At the end of each phase, the evolving software product is subject to a review process to try to uncover problems as soon as possible. A review might be either an *inspection* or a *walk-through,* without regard to the distinctions made in the software engineering

Table 2. Method or tool use.

METHOD OR TOOL	PERCENTAGE OF COMPANIES
High-level languages	100
On-line access	93
Reviews	73
Program design languages	63
Some formal methodology	41
Some test tools	27
Code auditors	18
Chief programmer team	7
Any formal verification	0
Formal requirements or specifications	0

literature.[4] Nearly everyone agrees that reviews work, and nearly everyone uses them, but the ways reviews are conducted differ greatly. Most agree that software projects can be routinely completed within time and budget constraints that only a few years ago could be managed only by luck and sweat. Reviews were instituted first 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 it 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: managerial control and 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 the time of valuable people. Lower level managers prefer to use reviews only when they think reviews are appropriate.

The technical success of the review process rests on the expertise and interest of the people conducting the review, not on the mechanism itself. The review process must be continually changed 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.

New and old companies differed considerably in their approaches to reviews. New companies were less committed to reviews, treating code reviews as training exercises for junior employees or as verification aids for particularly difficult modules. Since the newer companies did not have a large existing software base, they emphasized rapid development rather than maintenance. However, as companies grew and aged, accumulating software, reviews seemed to take on added significance as an important verification aid.

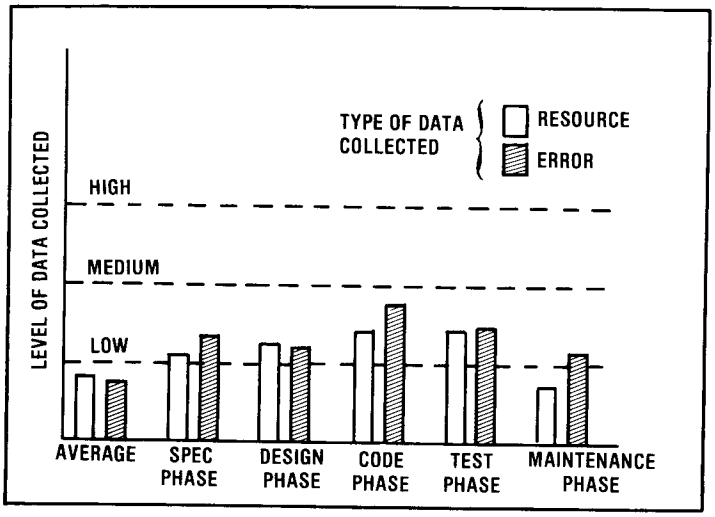


Figure 2. Amount and type of data collected.

Data collection. Every company collects some data, but not much of it becomes part of the corporate memory to be used beyond the project on which it was collected. Data generally belongs to individual managers, and they decide what to do with it. Data is rarely evaluated and used in an analysis to see if the process could have been improved. The opposite is true in Japanese companies in which "postmortem" analysis is frequently performed.

Several companies are experimenting with resource models, such as Price S. Slim.[5,6] No company seems to trust any model enough to use it extensively; instead the models are used to check manual estimates. Figure 2 shows that little data is being collected across all companies. The levels of Figure 2 are somewhat arbitrary; *high* represents the amount of data collected by the NASA Software Engineering Laboratory (resource use by programmer in each module, detailed error reports including causes, etc.), while *low* represents a minimal amount of data (e.g., the number of major and minor errors detected in reviews, outstanding error reports per time period, etc.).[7] In general, more error data than resource data is collected, and resource data is typically limited to hours spent by each programmer on the project to facilitate project billing.

It is extremely difficult to compare data across companies. First of all, quantitative data is quite rare within most companies. Error data is rarely tied to causes of errors, and the process of counting errors is never fed back into the development process. Knowing how many errors occurred does not necessarily improve the programming process if you don't know why they occurred. To keep the review process open, results are sometimes limited to the review group and the quality assurance manager. In addition each company has different definitions for most of the measured quantities, such as

- *lines of code,* which is defined as source lines, comment lines, with or without data declarations, executable lines or generated lines;
- *milestone dates,* which depend on the local software life cycle used by the company (whether requirements, specification, or maintenance data are included will significantly affect the results);
- *personnel,* which might include programmers, analysts, typists, librarians, or managers.

The differing definitions prevent any meaningful comparison. It is quite evident that the computer industry needs more work on the standardization of terms before it can address these quantitative issues. Also, it is not clear that management believes in a need for such data except for budgeting, so it is rarely collected.

Software development environments in the US

In describing the general characteristics of software environments, we limit discussion, for the most part, to the 13 companies in the US where we made site visits to over 20 different locations and interviewed approximately 60 project managers. Comments on Japanese software develop-

ment and comparisons with the techniques used in the US are presented in the subsequent section.

General life cycle. The life cycle of a project consists of the requirements and specification phase, the design phase, the code and unit test phase, and the integration test phase.

Requirements and specification phase. At all places contacted, requirements were in natural language text. Some projects had requirements documents that could be processed by machine, but tool support was limited to screen-oriented text editors. No analysis tools (like SREM and PSL/PSA were used except on toy projects.[8,9] Projects were either too small to justify the use of a processor or too large to make such use economical.

Reviews determine if the system architecture 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 depend greatly on the quality of people doing the review because the specifications are not formal. There is little traceability between specifications and designs.

Design phase. Most designs are expressed in some form of PDL or pseudocode, which makes design reviews effective. Tools that manipulate PDL vary from editors to simple text formatters. Only one company extended its PDL processor to analyze interfaces and the data flow in the resulting design.

While the use of PDL seems to be accepted practice, its use is not particularly effective. For example, we consider the expansion of PDL to code a reasonable measure of the detail in a design. A PDL-to-source-code expansion ratio of 1:1 may indicate that the design has the same detail as its eventual code. With design and code being separate tasks, this expansion indicates that the two concepts are not separated. The expansion ratios of PDL to source code were 1:5-10 at one location, 1:3-10 at another, and 1:1.5(!!)-3 at a third. Customer involvement with design varied greatly even within installations. Some produced volumes of PDL with an attitude similar to that for older projects that produced many detailed flowcharts: Nobody cares, but it's in the contract.

Code and unit test phase. Most code was in higher level languages—Fortran for scientific applications or some local variation of C, Pascal or PL/1 for systems work. In the aerospace industry, Fortran was the predominant language. People who normally worked in assembly language thought that Fortran and PL/1 signficantly 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 contains a concise review of this work.[10])

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, six to nine people, and on projects in which a person's responsibility was not divided among 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 does or doesn't believe the programmer. Almost no unit test tools are used to measure how effectively the tests devised by a programmer exercise the 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 ensure that such criteria have been met.

Integration test phase. 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. Testing in the systems software environment was not as rigorous compared to integration testing in the data processing environment.

Resources. Office space for programmers varied from one to two programmers sharing a Santa-Teresa style office with a terminal to large bullpens divided by low, movable partitions.[11] 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. Newer companies had two terminals per programmer and some were replacing terminals with personal computers. 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. A response time of 10 to 20 seconds was considered good at some places, where a subsecond response time was possible.[12]

Most companies are willing to invest in hardware, such as terminals, to assist their programmers but are reluctant to invest in software that might be as beneficial.

Education. Most companies have agreements with a local university to send employees for advanced training, and have their own seminar series. However, there is little training for project management. Only one company has a fairly extensive training policy for all software personnel.

Many companies had two problems with their educational program: (1) Programmers were sent to courses with

Table 3. Six lines of source code per staff-month.

LINES OF CODE	APPLICATION AND LANGUAGE
75	OS in Assembly
91	I/O controller in HLL
142-167	OS in HLL
182-280	Assembly applications

little or no follow-up experience, so what they learned was rarely put into practice and often forgotten; and (2) Some sites were far away from any quality university, and the isolation caused problems.

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. As Table 3 shows, the range in productivity was from 75 to 280 lines of code per month for different products using relatively similar development methods. This discrepancy illustrates that using lines of code as a measure of productivity is unwise and that more refined productivity measures are needed. Because of the differing application areas, we cannot really compare numbers in Table 3. However, it does seem obvious that the difficulty of the application area has more impact on productivity than the implementation language used (operating systems and other real-time programs being the most difficult).

One location reported that two major and five minor errors per 1000 lines were found during reviews in the design phase, and five major and eight minor errors per 1000 lines were found during reviews in the code phase. Realistically, though, the classification of errors into categories like *major* and *minor* may be useful for quality control in product distribution, but it sheds little light on the causes and possible treatments of these errors and their prevention in future systems.

US software developers are primarily producing applications, systems, or data processing software.

Development environments. The development environments centered on three types of projects: applications software, systems software, and data processing.

Applications software. We studied 13 projects in four 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 that smaller software houses could do whatever maintenance was necessary.

Since contracts are awarded after a competitive bidding cycle initiated by a "Request for Proposal," and requirements analysis is typically charged against company overhead, analysis is kept to a minimum before the contract is awarded. Requirements are written in English, and no formal tool is used. In addition, since the goal is to win a contract, there is a clear distinction between cost and price. *Cost* is the amount needed to build a product—a technical process that most companies feel capable of handling. On the other hand, *price* is a marketing strategy needed to win a bid. The price has to be low enough to win, but not so low that either money will be lost on the project or the company will be deemed "not responsive" to the requirements of the RFP. Thus, many ideas of software engineering developed during the 1970's on resource estimation and workload characterization are not meaningful in this environment because of 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 into products such as a communications network. In this case, most of the costs were for hardware, and software was 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 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 to 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.

Tool use was relatively sparse. Fortran was the dominant programming language. Two tools did seem to be used. In compliance with 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 programming tool that many companies did depend on, probably because the cost was low.

Staff turnover was uniformly low, generally five to 10 percent a year. Space for programmers seemed adequate, with one to two per office being typical. All locations except one used terminals for all computer access, and that one site had a pilot project to build private offices connected to a local minicomputer.

Systems software. We studied 18 projects produced by 11 vendors. Most of the projects were for large machines although some projects for microprocessors were included. Operating systems for those machines were the most important projects studied. The other projects, mostly compilers and utilities, did not follow the development rules for operating systems projects because the other projects were considered small, and hence their designs would be well understood.

Many companies are heading towards a policy of never building a large product. Development effort is limited to no more than two years and 10 programmers on any particular product. A great deal of effort is expended in the design of traditionally large pieces of software like operating systems to segment them into pieces of this size. Japanese companies also seem quite proficient at designing and assembling small projects only.

Software is generally written on hardware similar to the target machine. Terminals are universally used and the ratio of programmers to terminals varies from 1:2 to 3:1. Getting a terminal is frequently less of a problem than getting CPU cycles to do development.

In most places, software support is generally limited to text editors and interactive compilers. High-level development languages exist, and in most cases, the policy is that they be used; however, a substantial portion of operating systems remains in assembly language (20 to 90 percent depending on the company). The reasons are partly good (such as the prior existence of assembly code) and partly the usual: 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.

We studied 18 projects produced by 11 vendors. Most of these were for large machines, with operating systems being the major product.

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. One company assigned the quality control person directly to the development group, but group members believed that the independence of that individual in testing the system had been compromised as a result.

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

Programmers are usually organized into 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 percent per year) compared with that in the applications software area. Most programmers typically have private cubicles parceled 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. Not surprisingly, the older the system, the fewer software engineering techniques used.

Data processing. We studied seven data processing projects at five locations, although every location had some data processing activities for internal use. Most data processing software that we studied was developed in Cobol, although some systems were written in Fortran. There is a need to maintain the code throughout the life cycle.

Requirements were mostly in English and unstructured, although one company structured specifications by user function. Designs, especially for terminal-oriented products, were similar—a prototype 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 on how much the user was involved *before* 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 to the high level of interest on the part of the user assigned to the team during development.

All data processing was done at 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 percent per year. One location had a low turnover rate, which they attributed to their salaries being higher than those offered by comparable companies.

Data processing environments often use a phased approach to development, and quality control is especially important. One company, which had had numerous failures, attributed its 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.

Japan and US comparisons

Unlike the recent survey by Kim[2] which emphasized the integrated tool sets and artificial intelligence techniques being employed by Japanese industry, we found the level of technology used by the Japanese to be similar to US practices, but with some important differences. Programmers in both countries complain about the amount of money going toward hardware development and the lack of resources for software. However, we found that Japanese companies typically optimize resources across the company rather than within a single project. One effect of this policy is that tools become a capitalized investment paid for or developed out of company overhead rather than project funds. The cost of 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. Our survey does reinforce some of the conclusions of the earlier survey, namely:

(1) Japanese often use techniques developed in the US or Europe;
(2) Emphasis seems to be on practical tools.

Two successful techniques used by the Japanese are keeping projects small and relating failures to their causes through postmortem analyses of error data.

Improving software development in the US

While tool use is important for software development, the most important factor that we saw was the quality of the personnel on various projects. We firmly believe that tools cannot replace good people or good methods for software production. However, tool use *can* help a good programmer be even better.

Some very small changes can improve productivity in many installations. While there is 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) that supports these ideas.

- *More and better computer resources should be made available for development.* The computer systems being used for development are comparable to those available in the late 1960's or early 1970's providing 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.
- *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, which have formal languages and relatively straightforward compiler extensions. This organization could both acquire and evaluate the tools by looking at case studies and/or conducting experiments.
- *Tool support should be built for a common high-level language.* The tools we would pick first include a PDL processor, a syntax-directed editor—an editor that knows the underlying language syntax and prevents such errors from entering the program[13]—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 data flow 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 checks, such as those to verify whether begin-end blocks are aligned, are boring to perform manually and are therefore error prone. Unit testing tools can evaluate how thoroughly a program has been exercised. These tools are easy to build and should be accepted quickly, 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, such as design and source code; 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, and structural coverage metrics for test data.[14]
- *In syntax-directed editors, the grammar of the underlying language is built in, so the programmer needs to include only valid constructs at any point in the program.* Such systems facilitate top-down program development and often permit interactive context switching between program editing and program execution.[13,15] The programmer can then think more about algorithm design than keyboarding the program text.
- *The review process should be improved.* Reviews or inspections are a strong part of current methodology. The review process can be strengthened by the tools mentioned earlier. Manual labor in design and code reviews could be reduced with more effective tools. Such tools 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.
- *Incremental development such as iterative enhancement should be used.*[16] Many successful companies limited development to under two years and 10 people. One data processing company, after repeatedly failing to deliver software, decided never to build anything that required more than 18 staff-months. Since then they have been successful. Several other companies reported similar experiences. Large projects tend to have several layers of management and their success seems to depend on a stronger review process that comprises requirements, design, and code reviews. Smaller projects need less management and can succeed with only design reviews.
- *Data should be collected and analyzed.* 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 the analysis performed should be driven by a set of questions that need answers rather than by what is convenient to collect and analyze. For example, classifying errors into major and minor categories does not shed any light on development activities. A more detailed examination of error data can determine the causes of common errors, many of which may have remedies. Project postmortems should be conducted.

Observations

We identified several approaches for improving software productivity although they are not strictly supported

(or contradicted) by the data we collected. We offer these to stimulate discussion on this important topic.

- *Maintain compiler technology.* Many companies seem to contract out compiler development to smaller software houses because the nature of building most compilers is pedestrian. While compiler technology is relatively straightforward and perhaps cheaper to contract to a software house, the implications are far reaching. Software research is heading toward an integrated environment that covers the entire life cycle of software development. Research papers are being written about requirements and specification languages, design languages, program complexity measurement, knowledge-based Japanese fifth-generation languages, etc.[17] All of these depend on mundane compiler technology as their base.
- *Try prototyping.* Prototyping is one of the hot topics in software engineering literature today.[18] It is also crucial for all other engineering disciplines, but it was never mentioned on any survey form or during our visits. A form of prototyping in building data processing application software was common: the creation of screen displays during the design process. This technique and others should be expanded.
- *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 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.[19]
- *Examine the maintenance process.* Surprisingly, maintenance was rarely mentioned in our interview process, even though it is an expensive activity that most companies engage in. A Japanese project is 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.
- *Encourage innovation.* Experimental software development facilities are needed. Management should be encouraged to use new techniques on small funded-risk projects.

In preparing this article for publication, we were grateful for the thoughtful insights and comments from the reviewers. However, two issues kept on creeping into their reviews, and we suspect that the reader might have similar opinions.

(1) The comment was made that the reviewer (or his colleagues) used more or better tools, thus the survey was not representative. However, as we stated at the beginning, the goal was to look at production programmers—not research laboratories. We suspect that most reviewers and probably many readers of *Computer* also represent the research category.

(2) Names were sometimes given to demonstrate that industry is doing something about the problem. However, every person mentioned by the reviewers was interviewed by us and is included in this article. They either represent a research environment, and are not involved in "revenue-producing" software, or are considered an anomaly within their own company.

We are not saying here that software practices are dismal in the US. Technology transfer takes time, and it appears that the current level in industry represents the research environment of the mid-1970's—a delay of only 10 years. However, certain practices we mentioned do hinder technology transfer. We hope that this article is an impetus to address those issues so that discussions can start within companies to improve the process and shorten—still further—the time it will take to adopt good practices in industry. *

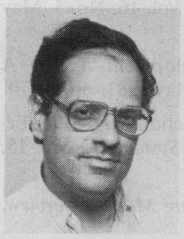
Acknowledgments

This project was sponsored by the IBM Corporation. We also acknowledge the cooperation of the following organizations for allowing us to survey their development activities: Bankers Trust Company, AT&T Bell Telephone Laboratories, Digital Equipment Corporation, Hewlett-Packard, Honeywell Large Information Systems Division, Kozo Keikaku Kenkyujo, Japan Information Processing Service, Microsoft, Nomura Computer Systems Ltd., Software Research Associates (Japan), Sperry Univac, System Development Corporation, Tandem Computers, Tokyo Electric Power Company, Toshiba Corporation, TRW, Wang Laboratories, and Xerox Corporation. Several Japanese companies did not respond to our request to use their names, so they are not listed here. This project would not have been possible without the help of all these companies.

References

1. *Software Engineering,* P. Naur and B. Randell, eds., NATO Scientific Affairs Division, Brussels, 1969.
2. K. H. Kim, "A Look at Japan's Development of Software Engineering Technology," *Computer,* Vol. 16, No. 5, May 1983. pp. 26-37.
3. R. C. Houghton, "Software Development Tools: A Profile," *Computer,* Vol. 16, No. 5, May 1983, pp. 63-70.
4. M. E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems J.,* Vol. 15, No. 3, Mar. 1976, pp. 182-211.
5. F. Freiman and Park, *PRICE Software Model Overview,* RCA, *Cherry Hill, N.J.,* Feb. 1979.
6. L. Putnam, "SLIM Software Life Cycle Management Estimating Model: User's Guide," *Quantitative Software Management,* (July 1979).
7. *Collected Papers: Volume 1,* tech. report 82-005, NASA Goddard Space Flight Center, Software Engineering Laboratory, Code 582.1, Greenbelt, Md., 1982.

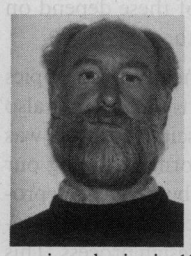
8. M. W. Alford, "A Requirements Engineering Methodology for Real-time Processing Requirements," *IEEE Trans. Software Engineering,* Vol. SE-3, No. 1, Jan. 1977, pp. 60-69.
9. D. Teichroew and E. A. Hersey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. Software Engineering,* Vol. SE-3, No. 1, Jan. 1977, pp. 41-48.
10. F. P. Brooks, Jr., *The Mythical Man-Month.,* Addison-Wesley, Reading, Mass., 1975.
11. G. M. McCue, "IBM's Santa Teresa Laboratory—Architectural Design for Program Development," *IBM Systems J.,* Vol. 17, No. 1, Jan. 1978, pp. 4-25.
12. A. J. Thadani, "Interactive User Productivity," *IBM Systems J.,* Vol. 20, No. 4, Apr. 1981, pp. 407-423.
13. T. Teitelbaum and T. Reps, "CPS—The Cornell Program Synthesizer," *Comm. ACM,* Vol. 24, No. 9, Sept. 1981, pp. 563-573.
14. V. R. Basili, *Models and Metrics for Software Management and Engineering,* IEEE-CS Press, Los Alamitos, Calif. 1980.
15. M. V. Zelkowitz, "A Small Contribution to Editing with a Syntax-Directed Editor," *Proc. ACM Sigsoft Symp. Practical Software Development Environments,* Apr. 1984, pp. 1-6.
16. V. R. Basili and A. J. Turner, "Iterative Enhancement: A Practical Technique for Software Development," *IEEE Trans. Software Engineering,* Vol. SE-1, No. 4, Dec. 1975, pp. 390-396.
17. H. Karatsu, "What Is Required of the Fifth Generation Computer—Social Needs and Its Impact," *Fifth Generation Computer Systems,* North-Holland, Amsterdam, 1982.
18. "ACM SIGSOFT Software Engineering Symposium: Workshop on Rapid Prototyping," *ACM Sigsoft Software Engineering Notes,* Vol. 7, No. 5, Dec. 1982.
19. A. Duncan and J. Hutchinson, "Using Attributed Grammars To Test Design and Implementation," *Proc. IEEE Fifth Int'l Conf. Software Engineering,* 1981, pp. 170-178.



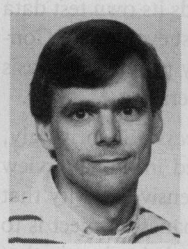
Martin V. Zelkowitz has been with the Computer Science Department of the University of Maryland since 1971 where he is an associate professor and, since August 1982, an associate chairman for education. His research interests include compiler and language design and the building of integrated tools for software development. He has also worked with the Institute for Computer Sciences and Technology of the National Bureau of Standards since 1976 and is the chairman of the Computer Society's Technical Committee on Software Engineering. He is past chairman of ACM Sigsoft and is now on the Sigsoft Executive Committee.

He received a BS in mathematics from Rensselaer Polytechnic Institute in 1967 and an MS and PhD in computer science from Cornell University in 1969 and 1971.

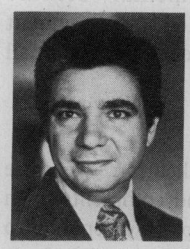
Raymond T. Yeh is professor of computer sciences at the University of Maryland. He has served as chairman of the Computer Science Departments at the Universities of Maryland and Texas at Austin. He was also director of the Center for Information Sciences Research at Maryland. Yeh received a BS in electrical engineering, an MA in mathematics, and a PhD in mathematics from the University of Illinois. He is the founding editor-in-chief of *IEEE Transactions on Software Engineering.*



Richard G. Hamlet has been with the University of Maryland since 1971 and is now associate professor. He will move to the Oregon Graduate Center in July 1984. His research interests include computability theory, programming languages, and software engineering, particularly testing theory. Hamlet received a BS in electrical engineering in 1959 from the University of Wisconsin in Madison, an MS in engineering physics in 1964 from Cornell University, and a PhD in computer science in 1971 from the University of Washington in Seattle.



John D. Gannon has been with the University of Maryland since 1975 and is now associate professor. His research interests include human-factors approaches to programming language design, formal specification and validation, and distributed computing. Gannon received an AB in mathematical economics in 1970 and an MS in computer science in 1972 from Brown University in Providence, R.I., and a PhD in computer science in 1975 from the University of Toronto.



Victor R. Basili is professor and chairman of the Computer Science Department at the University of Maryland in College Park, where he has been involved in the design and development of several software projects, including the SIMPL family of programming languages. He is currently measuring and evaluating software development in industrial settings and has consulted with many agencies and organizations, including IBM, GE, CSC, Naval Research Laboratory, Naval Surface Weapons Center, and NASA. Basili has published over 50 papers on the methodology, the quantitative analysis, and the evaluation of the software development process and product. In 1982, he received the Outstanding Paper Award from *IEEE Transactions on Software Engineering.* He serves on the editorial boards of the *Journal of Systems and Software* and *IEEE Transactions on Software Engineering.* He is a member of the ACM and the IEEE-CS Executive Committee of the Technical Committee on Software Engineering.