SOFTWARE ENGINEERING – AS IT IS

Barry W. Boehm
TRW Inc.
Redondo Beach, CA 90278

SUMMARY AND ABSTRACT

This paper presents a view of software engineering as it is in 1979. It discusses current software engineering practice with respect to lessons learned in the past few years, and concludes that the lessons are currently not heeded roughly half of the time. The paper discusses some of the factors which may account for this lag, including rapid technological change, education shortfalls, technology transfer inhibitions, resistance to disciplined methods, inappropriate role models, and a restricted view of software engineering.

The paper also updates a 1976 state of the art survey of software engineering technology, including such topics as requirements and specifications, design, programming, verification and validation, maintenance, software psychology, and software economics. It concludes that the field is making solid progress, but that it is growing more complex at a faster rate than we can put it in order.

SOME SOFTWARE ENGINEERING LESSONS LEARNED

Recently, I reviewed a paper which succinctly summarized many of the software engineering lessons we have (hopefully) learned over the past few years. Here are some excerpts:

1. Testable Requirements

"As soon as specifications for a system program are definitive, contractors should begin to consider how they will verify the program's meeting of the specifications. In fact, they should have had this in mind during the writing of the specifications, for it is easy to write specifications in such terms that conformance is impossible to demonstrate. For example: 'The program shall satisfactorily process all input traffic presented to it.' "

2. Precise Interface Specifications

"The exact interpretation of digital formats, the rise and fall times of waveforms, special restrictions as to when each type of data may or may not be sent – these and sundry other details must be agreed on by all parties concerned and clearly written down. Accomplishing this is apt to be a monumental and tedious chore, but every sheet of accurate interface definition is, quite literally, worth its weight in gold."

3. Early Planning and Specification

"If management takes the casual list-on-paper attitude toward a computer program, the consequence will be procrastination of complete program specification, followed by disbelief and consternation when lack of a proper program delays the whole system."

4. Lean Staffing in Early Phases

"The designers should not be saddled with the distracting burden of keeping subordinates profitably occupied. … Quantity is no substitute for quality; it will only make matters worse."

5. Core and Time Budgeting

"Budgets of time and storage, as mentioned earlier, should be set up, and monthly or more frequent reports are advisable on how well they are being adhered to. … [For storage budgets, include] … a safety factor of 25% or more, depending on the estimator's self-confidence and the likelihood of expansion in program requirements (they *always* expand)."

6. Careful Choice of Language

"Choosing a [Higher Order Language], like choosing a wife, is hard to undo after getting involved, and is not to be taken lightly."

7. Objective Progress Monitoring

"Percent-of-completion estimates will be asked for, and unless tasks are defined with unusual care, figures will be difficult to arrive at or decidedly misleading."

8. Defensive Programming

"Programmers should be imbued with the doctrine of anticipating possible troubles and detecting or correcting them [in their program]."

9. Integration Planning and Budgeting

"A common error in planning production of a program is to underestimate the time needed to combine units after they have been coded."

10. Early Test Planning

"Program acceptance tests should be defined early enough for contemplated acceptance-test inputs to be used in the terminal stages of program checkout."

HOW WELL HAVE THE LESSONS BEEN LEARNED?

Let us compare the above lessons learned with some samples of current software engineering practice gathered from a set of 50 term papers from a software engineering course I gave at USC earlier this year. The examples are drawn from recent government, industry, and university software projects in the Los Angeles area, and should form a reasonably representative sample of "Software Engineering, As It Is" as seen by the working-level software engineer.

1. Testable Requirements

"A requirements spec was generated. It has a number of untestable requirements, with phrases like 'appropriate response' all too common. The design review took weeks, yet still retained the untestable requirements."

"The only major piece of documentation written for the project was a so-called specification. Actually, the specification was written after the program was completed and looked more like a user's manual."

2. Precise Interface Specifications

"No one had kept proper control over interfaces, and the requirements specs were still inexact."

"The interface schematics were changed over the years and not updated, so when we interfaced with the lines, fuses were burned, lights went out…".

"The interface between the two programs was still not exact. When interfacing the two programs we ran into run time errors. Debugging was difficult because of the lack of documentation. We also began to forget exactly what our code did in certain situations and wished we had done more documentation."

3. Early Planning and Specification

"Despite one team member's efforts to develop a plan and some interface specs, the other two members felt there was no time or need to plan anything, and that each member should begin coding to complete the project on time. In fact, this did not save time, but caused many problems and delays."

"A software development plan was thrown together at the customer's request. It contained such good words as 'structured programming,' 'chief programmer team,' 'structured walkthroughs,' etc. This plan has been ignored since its creation, both by the project manager and the software head."

"This is all common sense, yet I know of no R&D minicomputer installation that uses a formal documentation procedure. It is with surprise that an engineer finds that paperwork can actually save time."

4. Lean Staffing in Early Phases

"At an early stage in the design, I was made the project manager and given three trainees to help out on the project. My biggest mistake was to burn up half of my time and the other senior designer's time trying to keep the trainees busy. As a result, we left big holes in the design which killed us in the end."

5. Core and Time Budgeting

"The core size is already three times the budget, and is running over the 90% mark. Two-thirds of the program is running from slow memory, making the execution time well over budget as well."

"This machine had a limited core size which resulted in much trickery and use of machine-dependent techniques in order to get the program to fit."

"Little planning was done, and the estimates of what the software development would entail were arbitrarily cut by the first project manager."

6. Careful Choice of HOL

"Although two other computer systems were clearly better as a host for our upgrade, we were locked into Brand X because of the huge inventory of code we had written in a Brand X-oriented HOL."

7. Objective Progress Monitoring

"Monthly status reports saying X% complete were given to the customer. As predicted in the text, the 50% mark tended to get reported as 90% complete."

8. Defensive Programming

"The programmer was a victim of the sad illusion that if the users were given a set of rules for entering the data, they would enter the data correctly. She had not even dreamed of the things users could do to destroy the database."

"The program is not very guarded. In an effort to save money, several types of error checking proposed were eliminated."

9. Integration Planning and Budgeting

"No integration plan has been constructed, nor have configuration management procedures been established."

10. Early Test Planning

"The acceptance test was a disaster. The users got into some of the exotic options and everything blew up. After that, we had a hard time getting them to believe anything we said about the system. There was no test planning – we just rushed into it blind."

11. Software Standards: General

"The design was not in modules, making it impossible to extend the use of the program. It was easier to write a new program than modify or correct the existing one."

"Rather than attempt to restructure the particular area that is being worked on, most of the programmers insert "patches" that cause the flow of control to snake around so that it is nearly impossible to try to follow the logic."

12. Software Management: General

"Morale dropped to such a low level that we were no longer a team. Deadlines were not met, interfaces did not work, programs did not fit requirements, and people quit."

To be fair, I should point out that the above excerpts are not a fully representative sample of the project experiences cited in the term papers. There were about an equal number of positive experiences in which people had evidently learned the lessons identified above and applied them successfully.

HOW SOON WILL WE LEARN THESE SOFTWARE ENGINEERING LESSONS?

Still, a 50% rate of applying these lessons is not very acceptable. It would certainly appear that the paper quoted above speaks directly to the current problems faced by software engineering practitioners. Its advice is timely, topical, and evidently much-needed today. It would appear, for example, to be a good candidate for the *Proceedings of this Conference*.

Unfortunately, the paper has already been published. In 1961.

It is "Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming," by W.A. Hosier. It appeared in the IRE Transactions on Engineering Management in June, 1961. It is based on the software engineering experiences accumulated on the U.S. SAGE and BMEWS command and control projects in the late 1950's. In the software engineering field, I would recommend everyone reading it, for two main reasons:

1. Although parts of the paper are a bit dated, most of its advice is still very good and well-expressed. (Perhaps the most striking anachronism in the world of 1979 is the statement that choosing a wife is hard to undo.)

2. I would hope that, in reading it, you would find yourself bothered, as I was, by the question:

   "If we knew all those things 18 years ago, why aren't we doing them now?"

I think this question is worth exploring, as its answers are certainly relevant to the topic question: "How soon will we learn these software engineering lessons?" In particular, the next section will discuss some likely reasons why our progress has been slow in assimilating software engineering techniques, and consider ways that we might be able to speed up the process.

SOME FACTORS INHIBITING GENERAL PROGRESS IN SOFTWARE ENGINEERING PRACTICE

Here are six factors which I believe are inhibiting our general progress in software engineering practice:

1. The field is growing rapidly.

2. We aren't teaching many of the above lessons to students.

3. Technology transfer is slow.

4. We resist the required discipline.

5. We have our role models mixed up.

6. We often take a restricted view of software engineering.

1. The Field Is Growing Rapidly

Our software engineering techniques have to be reexamined every time we are confronted with a significant change in the computer technology we work with. Although we generally find (as with microprocessors) that the general principles of software engineering still apply [Rauscher 78, Magers 78], we find that some techniques (e.g. instrumentation and test techniques) need to be modified to fit the new technology. There is not much that we can or want to do to eliminate this source of problems.

The field is also growing rapidly in terms of the number of people assimilated per year, who must necessarily relearn many software engineering lessons for themselves. We should be able to improve our status here via education, as discussed below.

2. We Aren't Teaching Many of the Above Lessons to Students

A recent survey [Thayer et al. 79] of software engineering instruction found large discrepancies between what professors felt were critical software engineering issues and what was being covered in software engineering courses. Of 20 major issues, only two (plan for maintainability and control quality) were covered to the extent commensurate with the issue's criticality. The other 18 issues (e.g., plan requirements, plan project, plan test, control visibility) have a very high correlation with the issues discussed above, but were under-covered in current courses.

The main reasons given for not covering those issues more were:

– lack of expertise
– lack of texts and other teaching materials
– inappropriate for computer science departments

Hopefully, recognition of the first two reasons will lead to activities to fill the needs expressed. The third reason was given by some professors who felt that such management-oriented material belonged in the business school. However, none of the business schools were teaching such material, either.

Our University education does best when it teaches people *fundamentals*: concepts and approaches which will serve the student through his entire practicing career. Speaking personally, I would have been helped much more in my overall software engineering career by a course on how to apply Hosier's lessons learned, or by a course on software economics, than I have been by course material I had on optimal sorting on a 2-tape machine with a poorly buffered memory, or on predicting the growth of truncation errors on machines with poorly designed arithmetic units, I'm not sure that today's microprocessor courses on how to hack around the problems in the Cromemco loader are much better from the standpoint of teaching fundamentals. Thus, I think that a reasonable argument can be made that the topics above are appropriate for computer science departments.

Good progress is being made by the IEEE/CS Subcommittee on Software Engineering Education toward defining a Master's curriculum in software engineering [Fairley 79]. However, progress in implementing the curriculum is slow. At this writing, only one University (TCU, under A. J. Hoffman) has, to my knowledge, actually established a formal Masters degree in software engineering.

3. Technology Transfer is Slow

Over the past few years, I have been able to observe some of the ways in which software engineering technology is transferred into a large industrial organization such as TRW. The "technology transfer measuring stick" shown in Figure 1 is one attempt to characterize what it is that makes such an organization adopt a new technology item. If a paper is published which simply presents an idea which addresses a significant problem, the idea may be picked up and used, but rarely. (Sometimes we have done this, only to find that even the originator had subsequently discarded the idea, although not as publicly as it was presented.) In general, software development people want to be assured that the idea has worked successfully in practice, on a job similar to theirs, and (ideally) by people who are available to apply it to their project.

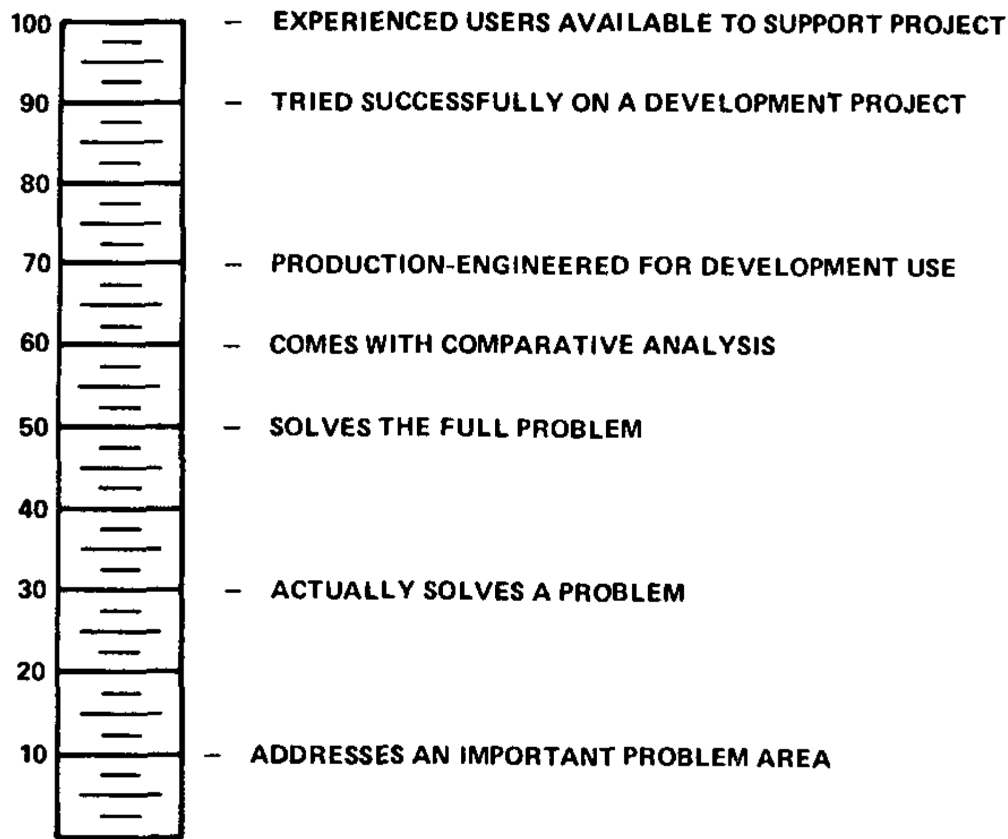

Figure 1. Research Results: A Measuring Stick for Successful Technology Transfer

This ideal is not always feasible, but in reviewing software engineering contributions which have been picked up most readily at TRW, they tend to cluster at the higher end of the measuring stick in Figure 1. Some examples are top-down structured programming [Baker 75], Pascal [Wirth 71], ISDOS [Teicheroew-Hershey 77], PDL [Caine-Gordon 75], Concurrent Pascal [Brinch-Hansen 77], and Parnas' specification and design techniques [Parnas 78, 79; Heninger et al. 78].

I have found that the best way to judge whether a paper is at the higher or lower end of the technology transfer scale is to look at its conclusions. If they say something like,

> "This technique has been implemented and found superior to other techniques for the following classes of problems: …"

then the contribution will tend to be at the top of the scale. If the conclusions say something like,

> "Although this technique has not been implemented, the author believes…"

or if the paper has no conclusions at all, then the contribution will tend to be at the bottom of the scale. We can all help to improve technology transfer by moving our contributions higher up the scale, even though it may mean publishing fewer papers.

4. <u>We Resist the Required Discipline</u>

We are beginning to accept the fact that there is "A Discipline of Programming" [Dijkstra 76] which requires us to accept constraints on our programming degrees of freedom in order to achieve a more reliable and well-understood product. We are reaching the point where we are willing to tie ourselves down by declaring in advance our variable types, weakest preconditions, and the like. But our free spirits still rebel at tieing ourselves down more fully by declaring in advance just what software we are going to build, how we are going to put it together, who is going to verify it and how, and what is the user going to do with it once he gets it. It's still much more attractive to jump in and start laying code. I'm afraid that this particular problem will be a long time in going away.

5. <u>We Have Our Role Models Mixed Up</u>

Another related factor inhibiting the progress of disciplined software engineering practice is something we call the "Wyatt Earp Syndrome." The "Wyatt Earp" is the indispensable programmer: the one who carries the critical program logic and design decisions around in his head, never documenting anything he does. When the inevitable crisis comes along, only he can save the situation, coming on like Wyatt Earp saving the town from the bad guys. All too often, the result is that the indispensable programmer is given a raise or a bonus, and becomes a hero or role model for other programmers in the organization. And in the process, the organization has become even more dependent on its Wyatt Earp than it was before.

The solution? Jerry Weinberg, a highly humanitarian person, has described it concisely: "If a programmer is indispensable, get rid of him as soon as possible" [Weinberg 71].

6. <u>We Often Take A Restricted View of Software Engineering</u>

The engineering of large software systems is as complex as any engineering ventures in history. Even judging conservatively, a 1,000,000-instruction software product has at least 10,000 component functions (assuming 100 instructions per function), each of which can be specified and developed in at least two different ways. Thus, even at this function level, there are $2^{10,000}$, or about $10^{3,000}$, combinations of function choices which the software engineer must sort out*.

When dealing with this level of complexity, it is absolutely necessary to simplify things to make them intellectually tractable. In doing so, we often take a restricted view of “software engineering” which equates it to “programming methodology,” and then proceed to tackle our programming problems.

Problems With the Restricted View

This restricted view is a good thing from the standpoint of clarifying our approach to programming tactics. However, software projects which have taken this restricted view exclusively, to the exclusion of the resource engineering and human relations aspects of software engineering, have unfortunately proceeded to relearn many of the lessons discussed above. One order-processing software project did a beautiful job of top-down, deductive structured programming, only to come to grief because it had not spent enough effort defining the proper “top” of the system. As a result, the overall order-processing system was more inefficient, error prone, and frustrating to use than even its cumbersome manual predecessor. Another well-structured, modular, hierarchical real-time system failed when it simply would not work in real time on the user’s workload – a fact that would have been evident by an early workload characterization and resource analysis of the system.

The need for using the broad rather than the restricted view of software engineering is also seen in Figure 2, which summarizes the problems in computer system acquisition found in 151 U.S. General Accounting Office audits [GAO 77]. Although technology factors are a significant source of acquisition problems, it is clear from Figure 2 that the major problem sources are more in the areas of acquisition planning and project control.

---

* One successful early software product JOSS [Shaw 64] was characterized as “I0,000 small.decisions, 99.9% of which were made correctly.”

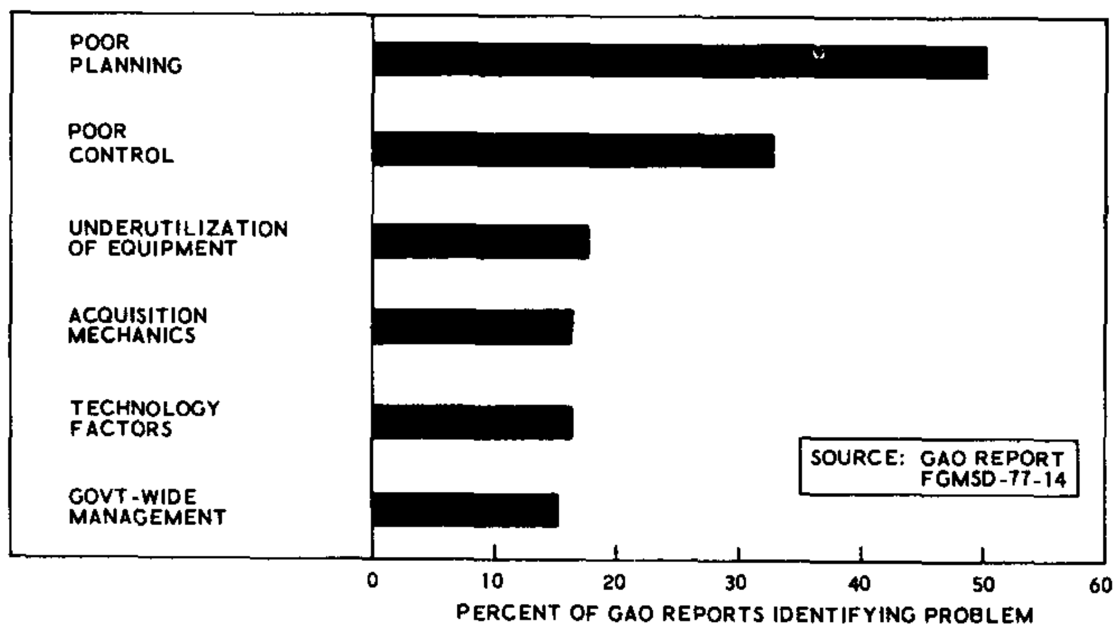

Figure 2. Problems with Computer System Acquisition and Use in U.S. Government, 1965–1976

Problems With the Sequential Approach

It would be convenient if we could perform a sequential "separation of concerns" which neatly factored the software engineering problem into a "programming-methodology" problem and a separable set of problems which covered all other considerations. There are two basic approaches for doing this – the "deductive top-down" approach and the "tuning" approach – but unfortunately they have not worked out well in general practice.

In the "deductive top-down" approach, all of the resource-engineering and human-relations problems are worked out in advance. The result of this activity is a fully-defined requirements specification, which can then be frozen and used as the basis of a deductive, top-down programming activity. Unfortunately, this approach will not work in general because of the fundamental volatility of software requirements. For example, IBM's Santa Teresa software organization has found, on a sample of roughly 1,000,000 instructions of software produced per year to IBM-determined requirements, that the average project experiences a 25% change in requirements during the period of its development [Climis 79].

The other approach is the "tuning" approach, in which the software is developed deductively from a first-cut set of requirements and then modified on the basis of user lessons learned in the meantime. Unfortunately, this approach will not work in general because of the inertia of user organizations. Once an initial software product is put into operation (however inappropriate), users tend to develop operational mechanisms to

compensate for the deficiencies in the software; these are extremely hard to change, once established. In these' cases, the degrees of freedom for tuning the system are much more restricted than had been originally anticipated, and a number of preferable modes of operation have become practically foreclosed by organizational inertia. In other cases, such as in some medical information systems, the initial programming product was so far from being acceptable to the doctors it was supposed to serve that the product was scrapped entirely, leaving no opportunity for tuning at all.

The Broad View of Software Engineering

For these reasons, it is important not to restrict our view of "software engineering" to cover only "programming methodology," but to adopt a definition of "software engineering" which encompasses the *necessarily concurrent* concerns of resource engineering and human relations. Fortunately, such a definition is consistent with common dictionary [Webster 79] definitions of "software" and "engineering."

- Software is the entire set of programs, procedures, and related documentation associated with a system and especially a computer system.

- Engineering is the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to man in structures, machines, products, systems, and processes.

Since the properties of matter and sources of energy over which software has control are embodied in the capabilities of computer equipment, we can combine the two definitions above as follows:

- Software Engineering is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures, and associated documentation.

Discussion

This definition of "software engineering" contains two key points which deserve further discussion. First, our definition of "software" includes a good deal more than just computer programs. Thus, learning to be a good software engineer means a good deal more than learning how to generate computer programs. It also involves learning the skills required to produce good documentation, data bases, and operational procedures for computer systems.

The second key point is the phrase "useful to man." From the stand-point of practice, this phrase places a responsibility upon us as software engineers to make sure that our software products are indeed useful to man. If we accept an arbitrary set of specifications and turn them into a correct computer program satisfying the specifications, we are not discharging our full responsibility as software engineers. We must also apply our skills and judgment, to the job of developing an appropriate set of

specifications, and to the job of ensuring that the resulting software does indeed make the computer equipment perform functions that are useful to man. Thus, concerns for the social implications of computer systems are part of the software engineer's job, and techniques for dealing with these concerns must be built into the software engineer's practical methodology, rather than being treated as a separate topic isolated from our day-to-day practice.

From the standpoint of learning, the phrase "useful to man" implies that the science and mathematics involved in software engineering covers a good deal more than basic computer science. For something to be useful to man, it must satisfy a human need at a cost that man can afford. Thus, the science and mathematics we must learn to apply as software engineers also includes the science of understanding human needs and human relations, i.e., psychology; the science of costs and values, i.e., economics; and the science of developing products within given cost budgets, i.e., management science.

Software Engineering Curriculum Implications

From the standpoint of learning, therefore, I find the current direction of evolution of the IEEE proposed masters curriculum in Software Engineering [Fairley 79] somewhat disappointing. The first draft of the curriculum contained both a course on Human Factors in Computing System Design and a course on Security and Privacy. In the second draft these were reduced to a single course which somewhat awkwardly tries to cover both topics together. Neither draft is very strong in the area of software engineering economics, although software management is appropriately highlighted in a course. Again, given the dominance of these concerns in practical software engineering situations, it is hoped that future iterations of the IEEE curriculum recommendations will contain a stronger emphasis on the broader view of software engineering.

RECENT DEVELOPMENTS

This portion of the paper will convey some of the recent developments in the field, as part of the "software engineering – as it is" charter of the paper. For brevity and convenience, I will only cover developments since the "Software Engineering" survey paper I wrote in 1976 [Boehm 76]. The first section will cover recent developments in the "programming methodology" areas of

1. Requirements and Specifications
2. Program Design
3. Programming
4. Verification and Validation
5. Maintenance

The second section will cover recent developments in the more broad-based areas of software engineering such as software phenomenology and economics, software psychology, and human factors in software engineering.

1. REQUIREMENTS AND SPECIFICATIONS

There are three main approaches for expressing what a software product is to do in a set of requirements specifications. These are:

1. Informal specifications. These are the traditional free-form natural-language specifications. They require virtually no training to write or read, but their ambiguity and lack of organization generally lead to serious problems with incompleteness, inconsistency, and misunderstandings among the various groups of people (users, buyers, developers, testers, trainers, interfacers, etc.) who must use them to guide their software development activities.

2. Formatted specifications. These are specifications expressed in a standardized syntax, which provides a framework for organizing the specifications and performing basic consistency and completeness checks. They generally require a moderate level of training to read and write well, but can be mastered easily by average programmers. Their formatted nature precludes certain sources of ambiguity, but their imprecise semantics implies that other sources of error are still present (e.g., one can define and use a term such as "mode" or "mechanism" and not pin down precisely what it means).

3. Formal specifications. These are specifications expressed in a precise mathematical form, with both syntax and semantics rigorously defined. They require a good deal more expertise and training to be able to read and write, and a longer time to write than formatted specifications. However, they eliminate virtually all sources of imprecision and ambiguity in a specification, and provide a basis of constructing a correct program and mathematically verifying its equivalence to the specification.

A good deal of progress has been made in the last three years in the development and use of both formatted and formal specifications. In the area of formatted specifications, the two major automated tools, ISDOS/CADSAT [Teichroew-Hershey 77] and SREM [Bell et al. 77, Alford 77] have continued to mature through use. ISDOS/CADSAT has added more powerful consistency and completeness checks, and a number of user-inspired improvements in data entry and output reports. SREM has become available on more host computers, has added capabilities to support business data processing and distributed processing applications, and has been used successfully by several organizations outside of its originators at TRW and the U.S. Army BMD

Advanced Technology Center [Alford 78]. Non-automated tests such as SADT [Ross-Schoman 77] have also experienced a significant expansion in the number and variety of successful users. In addition, other automated systems of formatted specification have been developed, such as AXES [Hamilton-Zeldin 79], a function-oriented specification method and SAMM [Lamb et al. 78], which appears to be strongly based on SADT. A good review of formatted specification techniques is found in [Ramamoorthy-So 78].

In the area of formal specifications, a good summary and discussion of recent progress can be found in [Liskov-Berzins 79] and in the discussion of this article by Parsons, Goguen, Hamilton, and Zeldin in [Wegner 79]. Most significantly, formal specification techniques are beginning to be used with success on nontrivial practical software products. Examples are the use of SPECIAL [Roubine-Robinson 77] in developing the (hopefully) same operating systems PSOS [Feiertag-Neumann 79] and KSOS [Berson-Barksdale 79], and the use of AFFIRM [Musser 79] in the attempt to verify the Delta military message-processing software [Gerhart-Wile 79].

Also, some significant progress has been made at bridging the gap between the more readable and easy-to-specify informal and formatted specifications, and the more precise, ambiguous formal specifications. One interesting approach is the Specification Acquisition from Experts (SAFE) approach in [Balzer et al. 77], which provides automated tools to help make an imprecise specification more precise. Another highly significant achievement is the large (over 400 pages) A-7 avionics software specification [Heninger et al. 78], [Heninger 79], based largely on the specification techniques of Parnas. This specification combines a great deal of precision with a great deal of readability, and handles the complexity of a large practical problem extremely well.

The main result of all this progress is to expand the domains in which formal and formatted specification techniques are practical to use. The best way I have found to characterize these domains is shown in Figure 3. It shows that the more expensive, time-consuming, expert-oriented formal methods are most appropriate when the requirements are very stable and don't need frequent rework (you can afford to use the formal methods) and when the application requires a very high degree of fault-freedom (you can't afford not to use the formal methods). The increased power, efficiency, and practical experience with formal and formatted specification techniques is pushing their domains of applicability down (more and more) toward problems with less stable software requirements and lower requirements for fault-freedom, continually reducing the domain in which informal techniques are preferable.

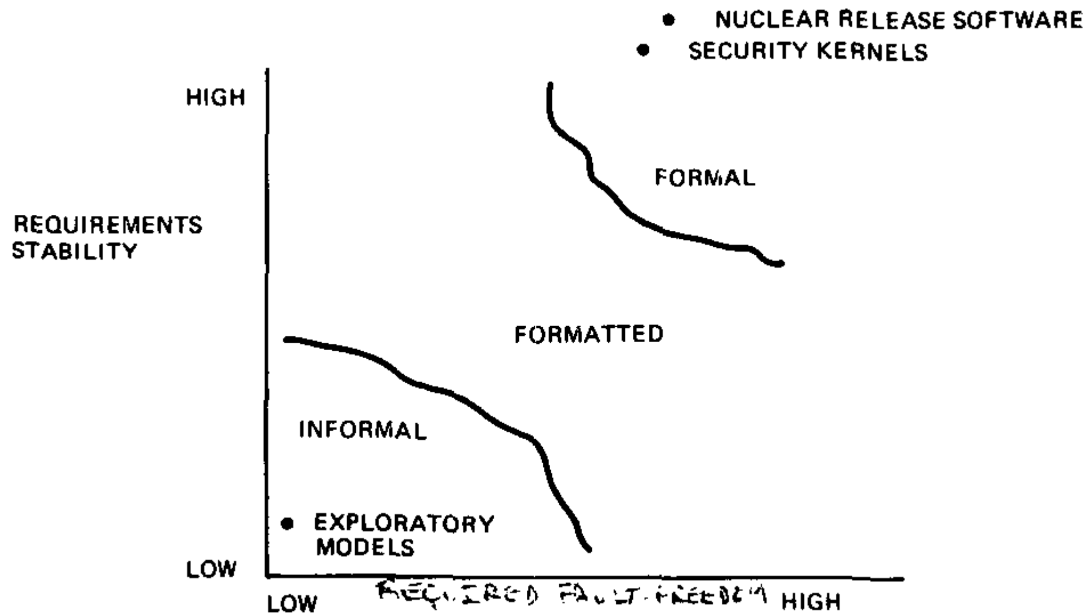

Figure 3. Problem Domains for Formal, Formatted, and Informal Methods

2. SOFTWARE DESIGN

A good deal of progress has also been made in the area of software design. Considerable experience with earlier design techniques has been accumulated, with the general conclusions that the HIPD technique has been less successful; that Program Design Language (PDL) [Caine-Gordon 75] has been successful and demonstrably superior to flowcharts [Ramsey et al. 78] ; and that the various forms of structured or composite design [Yourdon 75, Myers 75, Yourdon-Constantine 78, De Marco 78] have been highly successful.

Design techniques emphasizing data structuring have also been maturing, particularly those of [Jackson 75] and [Warnier 74]. The technology of life-cycle design took a significant step forward with the publication of Parnas' "Designing Software For Ease of Extension and Contraction" [Parnas 78, 79], which provides sound guidelines for using information-hiding principles to organize software in ways which make it easier to accommodate future changes. A good recent survey of software design techniques is [Freeman-Wasserman 77].

A number of organizations are developing extremely ambitious, all-encompassing computer-aided-design systems for software design, including capabilities for control structuring, data structuring, performance modeling, core budgeting, complexity analysis, assertions, flow analysis, cost analysis, and management tracking. These organizations would be well-advised to proceed with caution: such systems can easily collapse under their own weight. At least, this was our experience with the DEVISE system [Boehm et al. 75], which included ISDOS and PDL as subsets, along with a number of the

abovementioned capabilities. Although the originators were able to navigate reasonably well through the resulting welter of data entry requirements and control options, most other designers at TRW clearly preferred to use PDL (which was easy to learn and use) and imposed fewer constraints on their design approach. Our conclusion has been that a library of simple, limited-purpose design aids is a preferable way to proceed until we better understand the design process.

3. PROGRAMMING

The most significant recent advance in programming methodology has been the constructive approach to developing correct programs or "programming calculus" formulated in [Dijkstra 75], elaborated with numerous examples in [Dijkstra 76], and discussed further in [Gries 76]. This approach provides a clean, powerful method for working with a program specification to either derive a program structure which correctly implements the specification, or (just as important) to identify portions of the specification which are incomplete or inconsistent. At this point, it is becoming clear that the approach does not fully address all the problems involved in the development of large-scale software and user-oriented software (see the discussions by Gries, Homing, Liskov, and Parnas in [Wegner 79] and the discussion of "Problems with the Sequential Approach" above). However, the concepts involved provide powerful tools for attacking many of our most difficult programming problems.

A strong example of this power is given in the approach of [Hoare 78] in applying the concepts of weakest preconditions, guarded commands, and indeterminacy to provide a clear, disciplined set of techniques for dealing with the extremely difficult area of cooperating sequential processes. In the area of concurrent programming, a good deal of progress has also been made in applying the Concurrent PASCAL language to practical problems [Brinch Hansen 77, Brinch Hansen 78, Stepczyk 78] and determining improvements to cover some of the problems encountered with distributed monitors and with efficiency. Good surveys of other work in distributed and concurrent processing are given in [Stankovic-van Dam 79] and [Bryant-Dennis 79].

In the area of programming languages, the most significant development has been the progress toward the ADA language sponsored by the U.S. Department of Defense [Fisher 77, Ichbiah 79]. Although ADA has been widely criticized for its wide scope, ambitious incorporation of new concepts, and rapid timetable, it has come through so far as a well-designed, responsive language with a very good chance of becoming the next-generation standard programming language for a wide range of applications.

The amount of criticism directed toward ADA has been perhaps the most positive aspect of the entire ADA experience to date. No other language has had anywhere near as much open and broad-based review of its general and specific requirements and its preliminary and detailed design before proceeding into language development. In this respect, the ADA process provides a good model and base of experience for similar developments of highly standardized software in the future.

4. VERIFICATION AND VALIDATION

Progress in formal program verification (proof of correctness) techniques was discussed, to some extent, in the remarks above on formal specification techniques. Again, with respect to Figure 3, the current progress in formal methods implies that they already represent a viable option for application to practical problems with very stable requirements and a very high level of required fault freedom; and that their domain of applicability in Figure 3 will continue to expand. A survey of current work in formal verification techniques is given in [London 79].

Recent contributions have also given us a better understanding of what we should expect from formal verification techniques – which is considerably less than a full guarantee that the verified program is correct. A thorough analysis of a number of errors in published program proofs has been given in [Gerhart-Yelowitz 76]. An extensive discussion of other difficulties with formal verification – understandability and credibility of proofs, scaling up to large systems, program changes – and analogous experience in engineering and mathematics has been provided in [De Millo et al. 79], with the general conclusion that the practical use of formal techniques will never penetrate the lower regions of requirements stability and required fault-freedom in Figure 3.

Some progress has also been made on the theory of program testing, based largely on the concepts in [Goodenough-Gerhart 75]. Several papers by Hamlet, Richardson, and Ostrand and Weyuker in the recent IEEE Workshop on Software Testing and Test Documentation [Miller et al. 79] explored further the concepts of test validity and test reliability, with useful results in terms of improving test strategies. Some new approaches with both theoretical and practical interest have also been formulated: domain-testing [Cohen-White 77] and program mutation [De Millo et al. 78], A valuable discussion of these issues and techniques can be found in [Goodenough 79] and the counterpoint discussions of this article by Gerhart, Budd et al., and White et al. in [Wegner 79].

Considerable progress has been made in empirical studies of program testing and verification. A number of highly useful studies by Howden have established at least some initial results on the relative error-detection efficacy of various testing and analysis methods [Howden 77,78]. When these methods were tried on a common sample of programs containing 28 errors, they detected the following number of the 28 errors:

Path Testing –18
Branch Testing – 6
Structured Testing –12
Special Values – 17

Symbolic Execution – 17
Interface Analysis – 2
Anomaly Analysis – 4
Specification Requirements – 7

In the area of software reliability estimation, a valuable study of the predictive performance of several techniques has been performed by [Sukert 78]. This study showed that the predictions were generally not very close to the actual data, and also highly

sensitive to such features as the sampling interval and the time of sampling initiation. This behavior is most likely due to the failure of the independence assumption underlying the prediction models, as the actual error data was strongly conditioned by the sequence of test objectives pursued during the various test phases. Some much more successful predictions of software reliability have been obtained in [Musa '79]. One strong contributing factor is that the error data come from such contexts as trouble reports on a steady-state timesharing system, where the independence assumption is more valid.

Finally, two extremely helpful books have been published by Myers in the area of software testing and reliability [Myers 76, Myers 79], containing a great deal of useful practical guidance in achieving reliable software.

5. SOFTWARE MAINTENANCE

Thanks to a recent survey [Lientz-Swanson 78] of 487 business data processing installations, we now have a clearer picture of some of the gross characteristics of software maintenance. The survey confirmed that maintenance costs outweigh development costs: the percentages of total software effort break down as follows:

Development	:	43%
Maintenance	:	49%
Other	:	8%

Figure 4 shows how the maintenance effort is typically distributed. The largest component is due to updates (41.8%), but significant percentages are due to software repair (21.7%), and accommodating changes to input data and files (17.4%). This last activity thus consumes almost 9% of the total software budget, but it is difficult to identify much that is being done in the area of R&D to improve the process.

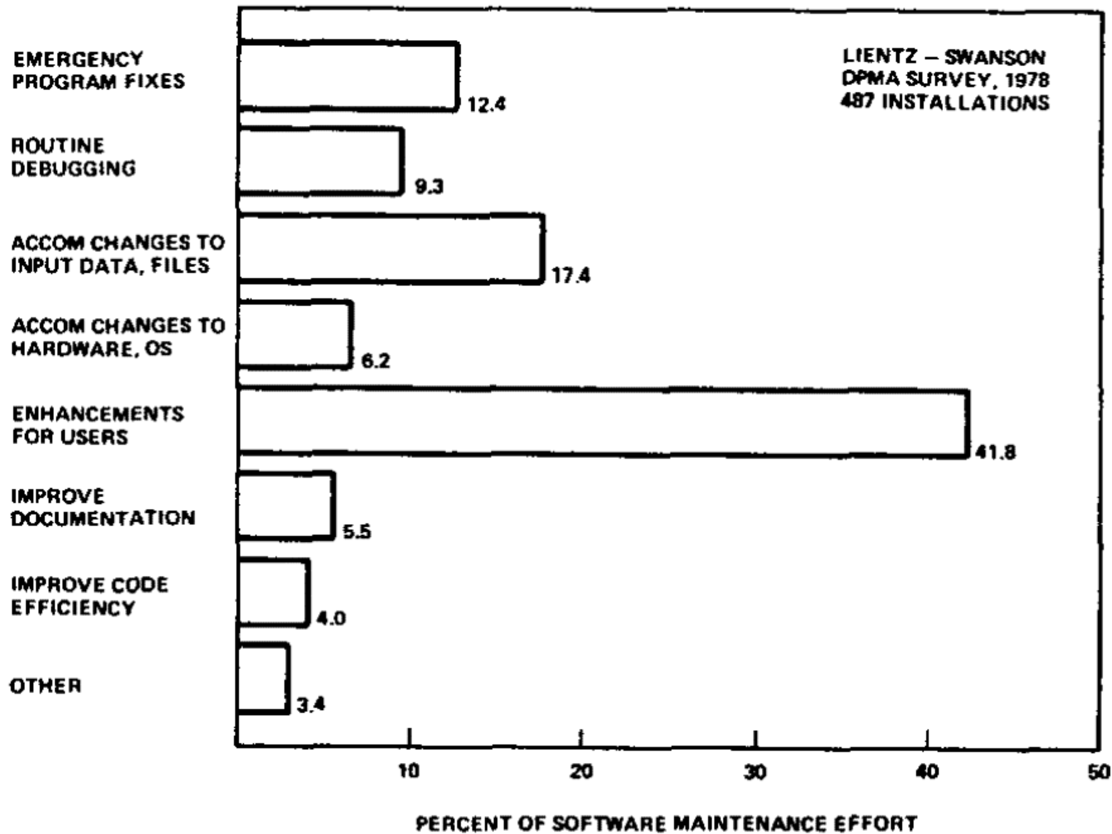

Figure 4. Distribution of Software Maintenance Effort

The most significant methodological advance to aid in designing for maintainability is the work on designing for extension and contraction in [Parnas 78, 79]. Some useful maintainability design checklists and standards are given in [Lipow et aI 77], and a good set of maintainability management guidelines is presented in [Munson 78]. Still, the software maintenance area is greatly underemphasized in current R&D efforts.

RECENT DEVELOPMENTS: INTEGRATED APPROACHES

This section of the paper discusses recent developments in areas which integrate program engineering concerns with other concerns such as human relations and software economics.

Software Psychology and Human Factors

The first major advance in software psychology was Weinberg's excellent book *The Psychology of Computer Programming* [Weinberg 71]. Subsequent work [Weinberg 72] established a significant correlation between components of programmer

performance and the objectives that programmers are given to optimize. More recently, some very useful insights into programmer motivation were obtained in [Couger-Zawacki 78], which showed that data processing personnel are significantly different from other classes of workers in the strength of their growth need and the weakness of their social need. (One moral: promoting top programmers into management is more likely to invoke the Peter Principle.) Other similar insights are given in [Fitz-Enz 78].

More and more useful studies are being performed on correlates of human performance in software situations, such as the complexity-measure experiments in [Sheppard et al. 79], the database query experiments in [Schneiderman 78], and the language experiments in [Gannon-Homing75]. Some very helpful studies and guidelines for engineering the software man-machine interface are discussed in [Dzida et al. 78] on interactive system design features, [Gilb-Weinberg 77] on humanized data entry via keyed input, and [Meister 76] on general man-machine task structuring. A good bibliography of progress in the field is [Atwood et al. 79]. An even better general reference will be the upcoming software psychology textbook [Schneiderman 79].

Software Phenomenology and Economics

The general quantitative study of software phenomenology has been the subject of two highly productive recent workshops [Lehman 77, Basili 78] covering such areas as software cost modeling; measurement of software reliability, complexity, and other qualities; quantitative software psychology studies; software maintenance phenomena; and general problems of software data collection and analysis.

One trend observable from these workshops and related studies is a significant degree of progress in the area of software cost estimation. Some examples of cost estimation models with improved predictive capability based on their calibration to at least 20 project data points each are the multiplicative Dory model [Herd et al. 77] and IBM-FSD model [Felix-Walston 77]; the Rayleigh curve-based Putnam model [Putnam 78]; and the composite RCA PRICE S model [Freiman-Park 79] and TRW SCEP model [Boehm-Wolverton 78]. Each of these efforts has resulted not just in a model but also in an increased understanding of the major factors influencing software costs, and their distribution across the various software life-cycle phases and activities.

Two examples of the latter are shown in Figures 5 and 6, which summarize the results of an experiment recently conducted by the author, involving the development of the same small software product (an interactive software cost estimation model) by two 5–6 person teams. Figure 5 shows the distribution of the resulting lines of code by function. The results for the two products are quite similar; in particular, note that only 2–3% of the code actually implements the model: the other 97–98% is devoted to various user interface, error handling, declaration, and other housekeeping functions. Figure 6 shows the distribution of project effort by activity, based on timesheets filled out by the participants. The distribution tends to reinforce the thesis that software engineering involves a good deal more than programming, which consumed only 9–10% of the project's effort.

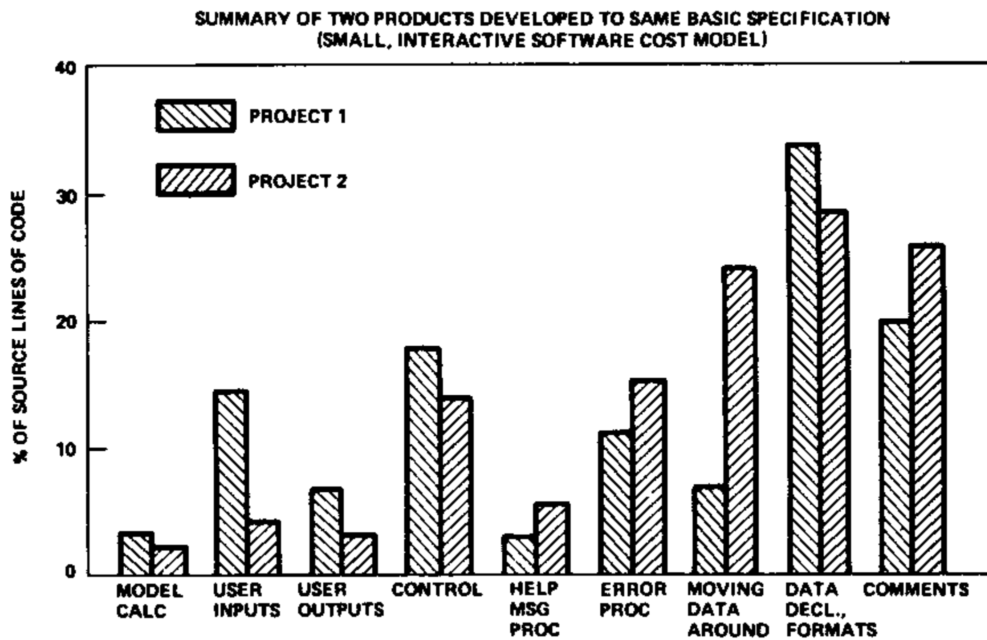

Figure 5. What Does a Software Product Do? Distribution of Source Code by Function

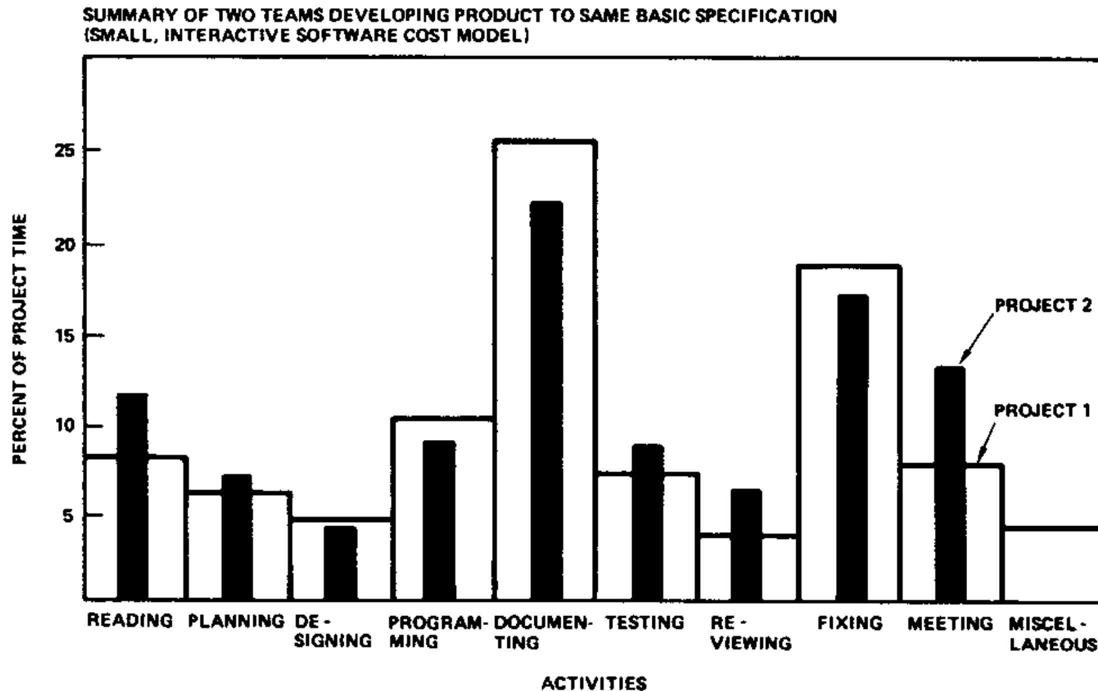

Figure 6. What Does a Software Project Do? Distribution of Project Time by Activity

Other significant contributions in this area include the analysis of RADC's large software data base of over 300 projects [Nelson 77] to obtain strong correlations between software project effort and schedule and software product size in number of instructions; the software evolution dynamics studies summarized in [Belady-Lehman 79]; and the encyclopedic analysis of general data processing costs and economics in [Phister 76].

CONCLUSIONS

In 1976, the software engineering field seemed to be preoccupied with what I called at the time *Area 1: detailed design and coding* of *systems software* by *experts* in a relatively *economics-independent* context; while the most pressing problems seemed to be in *Area 2: requirements analysis, design, test, and maintenance* of *applications software* by *technicians* in an *economics-driven* context. It is a real pleasure to observe that in 1979 there is not only a good deal more effort devoted to Area 2, but also that the effort is yielding highly useful and solid results.

Other signs of the increased maturity and sophistication of the field are the more careful qualifications and distinctions made in presenting and analyzing software engineering techniques today. A few years ago, the field seemed to abound with simplistic panaceas such as:

- “Proof techniques will guarantee reliable software.”
- “Put the processing on a chip and the software problem will go away.”
- “More detailed standards and procedures will make everybody good software managers.”
- “Automatic programming is just around the corner, and programmers will all be out of jobs.”
- “Eliminating GOTO’s will reduce your software budget by 50%.”

Nowadays, these oversimplified statements are heard very rarely, and there is much more emphasis on establishing the particular problem domains in which a given technique is effective or preferable to others. This has forced us to accept the fact that the software field is not a simple one, and that if anything, it is getting more complex at a faster rate than we can put it in order. But I suspect that most of us would agree with Bill Wulf’s assessmentof this trend [Wulf 79]:

> “The research trends described]here] will undoubtedly improve the situation, but history suggests that our aspirations will grow at least as fast as the technology to satisfy them. I, for one, would not want it any other way.”

REFERENCES

[Alford 77]. M. W. Alford, “A Requirements Engineering Methodology for Real-Time Processing Requirements,” *IEEE Trans. Software Engr.*, Jan. 1977, pp. 60–68.

[Alford 78]. M. W. Alford, “Software Requirements Engineering Methodology (SREM) at the Age of Two,” *Proceeding, COMPSAC 78*, IEEE, Nov. 1978, pp. 332–339.

[Atwood et al.79]. M. E. Atwood et al., “Annotated Bibliography on Human Factors in Software Development” U.S. Army ARI Technical Report P-79-1, Jun. 1979.

[Baker 75]. F. T. Baker, “Structured Programming in a Production Programming Environment,” *IEEE Trans. Software Engr.*, Jun. 1975, pp. 241–253.

[Balzer et al. 77]. R. Balzer, N. Goldman, and D. Wile, “The Inference of Domain Structure from Informal Process Descriptions,” *ACM SIGART Newsletter*, Jun. 1977.

[Basili 78]. V. R. Basil (ed), *Proceedings, Second Software Life-Cycle Management Workshop*, IEEE Catalog No. 78CH 1390-4C, Aug. 1978.

[Belady-Lehman 79]. L. A. Belady and M. M. Lehman, "Characteristics of Large Systems," in [Wagner 79].

[Bell et al. 77]. T. E. Bell, D. C. Bixler, and M. E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," *IEEE Trans. Software Engr.*, Jan. 1977, pp. 49–59.

[Berson-Barksdale 79]. T. A. Berson and G. L. Barksdale, Jr., "KSOS-Development Methodology for a Secure Operating System," *Proceedings, 1979 NCC*, pp. 365–371.

[Boehm et al. 75]. B. W. Boehm, R. L. McClean, and D. B. Urfrig, "Some Experience with Automated Aids to the Design of Large Scale Reliable Software," *IEEE Trans. Software Engr.*, Mar. 1975, pp. 125–133.

[Boehm 76]. B. W. Boehm, "Software Engineering," *IEEE Trans Computers*, Dec. 1976, pp. 1226–1241.

[Boehm-Wolverton 78]. B.W. Boehm, and R. W. Wolverton, "Software Cost Modeling: Some Lessons Learned," in [Basili 78].

[Brinch Hansen 77]. P. Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

[Brinch Hansen 78]. P. Brinch Hansen, "Network: A Multiprocessor Program," *IEEE Trans. Software Engr.*, May 1978, pp. 194–198.

[Bryant-Dennis 79]. R. E. Bryant and J. B. Dennis, "Concurrent Programming," in P. Wagner (ed), *Research Directions in Software Technology*, MIT Press, 1979.

[Caine-Gordon 75]. S.H. Caine and E.K. Gordon, "PDL: A Tool for Software Design," *Proceedings, 1975 NCC*, pp. 271–276.

[Climis 79]. T. Climis, "Software Cost Estimation," presentation at NSIA Software Workshop, Buena Park, CA, Feb. 1979.

[Cohen-White 77]. E. I. Cohen and L. J. White, "A Finite Domain-Testing Strategy for Computer Program Testing," Ohio State Univ. TR-77-13, Aug. 1977.

[Couger-Zawacki 78]. J. D. Cougar and R. A. Zawacki, "What Motivates DP Professionals?" *Datamation*, Sap. 1978, pp. 116–123.

[DeMarco 78]. T. DeMarco, *Structured Analysis and System Specification*, Yourdon Press,New York, 1978.

[De.Millo et al. 78]. R. A. DeMillo, R. J. Lipton, and F. E. Sayward, “Hints on Test Data Selection: Help for the Practicing Programmer,” *Computer*, Apr. 1978, pp. 34–41.

[DeMillo et al. 79]. R. A. DeMillo, R.J. Lipton, and A. J. Perils, “Social Processes and Proofs of Theorems and Programs,” *Comm. ACM*, May 1979, pp. 271–280.

[Dijkstra 75]. E. W. Dijkstra, “Guarded Commands, Nondeterminacy, and Formal Derivation of Programs,” *Carom ACM*, Aug. 1975.

[Dijkstra 76]. E.W. Dijkstra, A Discipline of Programming, Prentice-Hall Inc., Englewood Cliffs, NJ, 1976.

[Dzida et al. 78]. W. Dzida, S. Herda, and W. D. Itzfeldt, “User-Perceived Quality of Interactive Systems,” *IEEE Trans. Software Engr.*, Jul. 1978, pp. 270–275.

[Fairley 79]. R. E. Fairley, “MSE-79: A Recommended Masters Curriculum in Software Engineering,” Colorado State Univ., Feb. 1979.

[Feiertag-Neumann 79]. R. J. Feiertag and P. G. Neumann, “The Foundations of a Probably Secure Operating System (PSOS),” *Proceedings, 1979 NCC*, pp. 329–334.

[Felix-Walston 77]. C. P. Felix and C. E. Walston, “A Method of Programming Measurement and Estimation,” IBM Sys J. Vol. 16, No. 1, 1977.

[Fisher 77]. D.A. Fisher, “The Common Programming Language of the Department of Defense,” *Proceedings, AIAA/NASA/IEEE/ACM Computers in Aerospace Conference*, Oct. 1977, pp. 297–307.

[Fitz-Enz 78]. J. Fitz-Enz, “Who Is the DP Professional?” *Datamation*, Sap. 1978, pp. 124–129.

[Freeman-Wasserman 77]. P. Freeman and A.I. Wasserman, *Tutorial on Software Design Techniques* (2nd ed), IEEE Catalog No. 76CH1145-2-C, 1977.

[Freiman-Park 79]. F. R. Freiman and R. E. Park, “The PRICE Software Cost Model,” RCA Price Systems, Cherry Hill, NJ, Feb. 1979.

[Gannon-Horning 75]. J. D. Gannon, and J. J. Homing, “Language Design for Programming Reliability,” *IEEE Trans. Software Engr.*, Jun. 1975, pp. 179–191.

[GAO77]. U.S. General Accounting Office, “Problems Found with Government Acquisition and Use of Computers from November 1965 to December 1976,” GAO, Washington, DC, Report FGMSD-77-14, Mar. 1977.

[Gerhart-Yelowitz 76]. S. L. Gerhart and L. Yelowitz, "Observations of Fallibility in Applications of Modern Programming Methodologies," *IEEE Trans. Software Engr.*, Sep. 1976, pp. 195–207.

[Gerhart-Wile 79]. S. L. Gerhart and D. S. Wile, "Preliminary Report on the Delta Experiment: Specification and Verification of a Multiple User File Updating Module," *Proceedings, Specifications of Reliable Software Conference*, IEEE, Mar. 1979, pp. 198–211.

[Gilb-Weinberg 77]. T. Gilb and G. M. Weinberg, *Humanized Input*, Winthrop, Inc., Cambridge MA, 1977.

[Goodenough-Gerhart 75]. J. B. Goodenough and S. L. Gerhart, "Toward a Theory of Test Data Selection," *IEEE Trans. Software Engr.*, Jun. 1975, pp. 156–173.

[Goodenough 79]. J. B. Goodenough, "A Survey of Program Testing Issues," in P. Wegner (ed), *Research Directions in Software Technology*, MIT Press, 1979.

[Gries 76]. D. Gries, "An Illustration of Current Ideas on the Derivation of Correctness Proofs and Correct Programs," *IEEE Trans. Software Engr.*, Dec. 1976, pp. 238–243.

[Hamilton-Zeldin79]. "The Relationship of Design and Verification," *Journal of Systemsand Software*, Vol. 1, No. 1, 1979.

[Herd et al. 77]. J. R. Herd et al., "Software Cost Estimation Study – Study Results, RADC-TR-77-220, Vol. I, Jun. 1977.

[Heninger et al. 78]. K. Heninger, J. Kallander, D. L. Parnas, and J. Shore, *Software Requirements for the A-7E Aircraft*, Naval Research Laboratory Report 3876, Nov. 1978.

[Heninger 79]. K. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *Proceedings, Specifications of Reliable Software Conference*, IEEE, Mar. 1979, pp. 1–14.

[Hoare 78]. C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM*, Aug. 1978, pp. 666–677.

[Hosier 61]. W. A. Hosier, "Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming," IRE Transactions on Engineering Management, Jun. 1961, pp. 99–115.

[Howden 77]. W. E. Howden, "Symbolic Testing – Design Techniques, Costs, and Effectiveness," NBS Report GR 77-89, NTIS No. PB268517, 1977.

[Howden 78]. W. E. Howden, “Theoretical and Empirical Studies of Program Testing,” *IEEE Trans. Software Engr.*, Jul. 1978, pp. 293–298.

[Ichbiah et al. 79]. J. D. Ichbiah et al., “Rationale for the Design of the ADA Programming Language” and “Preliminary ADA Reference Manual,” ACM SIGPLAN Notices, Jun. 1979.

[Jackson 75]. M.A. Jackson, *Principles of Program Design*, Academic Press, 1975.

[Lamb et al. 78]. S. S. Lamb, V.G. Lack, L.J. Peters, and G. L. Smith, “SAMM: A Modeling Tool for Requirements and Design Specification,” *Proceedings, COMPSAC 78*, IEEE, Nov. 1978, pp. 48–53.

[Lehman 77]. M. M. Lehman (ed), *Software Phenomenology. Proceedings, U.S. Army Software Life Cycle Management Workshop*, Aug. 1977.

[Lientz-Swanson 79]. B. P. Lientz and E. B. Swanson, “Software Maintenance: A User/Management Tug-of-War,” *Data Management*, Apr. 1979, pp. 26–30.

[Lipow et al. 77]. M. Lipow, B. B. White, and B.W. Boehm, “Software Quality Assurance: An Acquisition Guidebook,” TRW-SS-77-07, Nov. 1977.

[Liskov-Berzins 79]. B. H. Liskov and V. Berzins, “An Appraisal of Program Specifications,” ;n P. Wagner (ed), *Research Directions in Software Technology*, MIT Press, Cambridge, MA, 1979.

[London 79]. R. L. London, “Program Verification,” in P. Wagner (ed), *Research Directions in Software Technology*, MIT Press, 1979.

[Mages 78]. C. S. Magers, “Managing Software Development in Micro-processor Projects,” *Computer*, Jun. 1978, pp. 34–42.

[Meister 76]. D. Meister, *Behavioral Foundations of System Development*, John Wiley and Sons, New York, 1976.

[Miller et al. 79]. E. Miller et al., “Workshop Report: Software Testing and Test Documentation,” *Computer*, Mar. 1979, pp. 98–107.

[Munson 78]. J. B. Munson, “Software Maintainability: A Practical Concern for Life-Cycle Costs,” Proceedings, COMPSAC 78, Nov. 1978, pp. 54–59.

[Musa 79]. J. D. Musa, “Software Reliability MeasuresApplied to System Engineering,” *Proceedings, 1979 NCC*, pp. 941–946.

[Musser79]. D. R. Musser, “Abstract Data Type Specifications in the Affirm System,” *Proceedings, Specifications of Reliable Software Conference*, IEEE, Mar. 1979, pp. 47–57.

[Myers 75]. G. J. Myers, *Reliable Software Through Composite Design*, Petrocelli-Chanter, 1975.

[Myers76]. G. J. Myers, *Software Reliability*, John Wiley and Sons, New York, 1976.

[Myers 79]. G. J. Myers, *The Art of Software Testing*, John Wiley and Sons, New York, 1979.

[Nelson 77]. R. Nelson, “Software Data Collection and Analysis at RADC,” Rome, NY, 1977.

[Parnas 78,79]. D. L. Parnas, “Designing Software for Ease of Extension and Contraction,” *Proceedings, ICSE3*, May 1978, pp ‘264–277, and *IEEE Trans. Software Engr.*, Mar. 1979, pp 128–137.

[Phister 76]. M. Phister Jr., *Data Processing Technology and Economics*, Santa Monica Publishing Co., 1976.

[Putnam 78]. L. H. Putnam, “A General Empirical Solution to the Macro Software Sizing and Estimating Problem,” *IEEE Trans. Software Engr.*, Jul. 1978, pp. 345–361.

[Ramamoorthy-So 78]. C.V. Ramamoorthy and H.H. So, “Software Requirements and Specifications: Status and Perspectives,” in C. V. Ramamoorthy and R. T. Yeh, *Tutorial: Software Methodology*, IEEE Catalog No. EHO 142-0, 1978, pp. 43–164.

[Ramsay et al. 78]. H. R. Ramsay,M. E. Atwood, and J. R. Van Doren, “A Comparative Study of Flowcharts and Program Design Languagesfor the Detailed Procedural Specification of Computer Programs,” U.S. Army ARI Technical Report TR-78-A22, Sep. 1978.

[Rauscher 78]. T. G. Rauscher, “A Unified Approach to Microcomputer Software Development,” *Computer*, Jun. 1978, pp. 44–54.

[Ross-Schoman 77]. D.T. Rossand K. E. Schoman Jr., “Structured Analysis for Requirements Definition,” *IEEE Trans. Software Engr.*, Jan. 1977, pp 6–15.

[Roubine-Robinson 77]. O. Roubine and L. Robinson, *SPECIAL Reference Manual*, SRI International, Menlo Park, CA, Jan. 1977.

[Shaw 64]. J.C. Shaw, “JOSS: A Designer’s View of An Experimental On-Line Computing System,” *Proceedings*, 1964 FJCC, pp. 455–464.

[Sheppard et al. 79]. S.B. Sheppard, B. Curtis, P. Milliman, M. A. Borst, and T. Love, "First-Year Resultsfrom a ResearchProgramOn Human Factors in Software Engineering," *Proceedings, 1979 NCC*, pp. 1021–1028.

[Shneiderman 78]. B. Shneiderman, "Improving the Human Factors Aspect of Database Interactions" *ACM Trans. Database Systems*, Dec. 178, pp. 417–439.

[Shneiderman 79]. B. Shneiderman, *Software Psychology*, Prentice-Hall, Inc., EnglewoodCliffs, NJ, 1979 (to appear).

[Stankovic-van Dam 79].]. A. Stankovic and A. van Dam, "Research Directions in (Cooperative) Distributed Processing," in P. Wagner (ed), *Research Directions in Software Technology*, MIT Press, 1979.

[Stepczyk78]. F. Stepczyk, "A Case Study in Real-Time Distributed Processing Design," *Proceedings, COMPSAC 78*, Nov. 1978, pp. 514–519.

[Sukert 78]. A. N. Sukert, "A Four-Project Empirical Study of Software Error Prediction Models," *Proceedings, COMPSAC 78*, Nov. 1978, pp. 577–582

[Teichroew-Hershey 77]. D. Teichroew and E. A. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. Software Engr.*, Jan. 1977, pp. 41–48.

[Thayer et al. 79]. R. H. Thayer, A. Pyster, and R. C. Wood, "Major Issues in Software Engineering Project Management," Sacramento Air Logisitics Center, May 1979.

[Warnier74]. 1. D. Warnier, *Logical Construction of Programs*, Van Nostrand Reinhold, New York, 1974.

[Webster 79]. *Webster's New Collegiate Dictionary*, G&C Merriam Co., 1979.

[Wegner 79]. P. Wagner (ed), *Research Directions in Software Technology*, MIT Press, Cambridge, MA, 1979.

[Weinberg 71]. G. M. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, 1971.

[Weinberg72]. G. M. Weinberg, "The Psychology of Improved Programming Performance," *Datamation*, Nov. 1972.

[Wirth 71]. N. Wirth, "The Programming Language Pascal," *Acta Informatica*, 1971, pp. 35–63.

[Wulf 79]. W. A. Wulf, "Comments on Current Practice," in [Wagner 79].

[Yourdon 75]. E. Yourdon, *Techniques of Program Structure and Design*, Prentice-Hall, 1975.

[Yourdon-Constantine 78]. E. Yourdon and L. L. Constantine, *Structured Design*, (2nd ed), Yourdon Press, New York, 1978.