Software engineering: An emerging discipline

by R. Goldberg

Software engineering is an emerging discipline whose goal is to produce reliable software products in a cost-effective manner. This discipline is evolving rapidly as the challenges faced by its practitioners keep extending their skills. This paper gives a quick tour of the main ideas and thrusts that have driven software engineering in its first 25 years and attempts to look ahead at the next set of advances.

Our society has adopted the use of computers at a rate that staggers the imagination. Every segment of society has discovered programs that seem to make its jobs more productive. This demand for programs has required a tremendous amount of code to be written by a population of producers whose training and tradition are recent. Every estimate of the number of programs that remain to be written, because of a perceived need that these programs are expected to fill, seems to depend on the number of programmers available to write the programs. Thus, when we increase the number of programmers, we also add to their work. However, the number of programmers appears limited.

The results of this dilemma have been diverse. They range from increased pressure on the existing programmer population to produce more in a shorter period, to attempts to change the way programs are produced through the invention of new tools and approaches.

At one extreme, the pressure to produce more with less has been one of the causes of disaffection with the software community of the 1960s and 1970s. There were many examples of attempts to produce software applications that had tremendous cost overruns, failed to provide the function promised, and were subject to unexpected and disastrous failure. The rising cry was, "Why isn't software production as predictable as engineering?" The response, at least partially, was to attempt to create a software engineering discipline to solve these problems.

At the other extreme, an attempt was made to find scientific principles leading to new insights and relationships that would revolutionize the way software is produced. This approach would certainly produce a better product. The direction in which to go stemmed from this approach and drew from what appeared to be the source discipline of software production, computer science. The faculties of colleges and universities that had computer science departments were asked to provide the new approaches, tools, or techniques to solve the problem of improving software development.

The requests for instantaneous fixes have to be viewed in the context of the past 25 years, during which time computer applications came to be accepted. In some ways the data processing community is being asked to ease the transfer of a new technology to the population at large. The focus of this rapidly changing technology is on its application to providing practical software to all segments of society. The model provided by other engineering disciplines should be useful in understanding the pace of development and the limitations to be faced by software

engineering. As an illustrative example, consider a stylized view of the growth of civil engineering as seen through the development of pyramids in Egypt.

The pyramids were monuments built by the pharaohs of Egypt as burial sites. A pharaoh attempted to have his name last through the centuries by building a structure of a size sufficient to ensure its indestructibility. The first pyramids must have been built by trial and error, until a successful approach was found. The skills so painfully learned by generations of builders were then passed on by apprenticing their children, relatives, and others.

As time went by, each successive pharaoh wanted a bigger pyramid built, and each successive generation of builders attempted to build it. But increasing the size was not all that simple. The relationship between the height of the pyramid and the area of the base required to support it was not linear, and the builders needed to learn the proper relationship. Egypt is dotted with collapsed pyramids attesting to lessons painfully learned.

Eventually this relationship was learned, and the pyramid builders developed rules of thumb to assist them in designing stable pyramids. The engineers, however, did not understand why this stability was attained and how it depended on the weight of the material with which the pyramid was constructed. They perhaps did not realize how different construction techniques would change the weight of the structure and allow a different relationship among the weight, area, and height. They might not have realized how using newer, lighter materials would have similarly altered these relationships. These insights came much later, after the tools of mathematical analysis and experimentation were applied by natural scientists to develop tables of rules of thumb that civil engineers could use for practical purposes.

This sequence of events implies the following stages: First, a pragmatic need causes trial-and-error attempts. The knowledge gained is not acquired systematically, and it is passed down from generation to generation and from master to apprentice. Second, a science is created to try to understand the rules of thumb that the "practical" engineers have discovered. In this process the scientists begin to put together a base of knowledge from which the engineer can learn in order to make the final product more predictable.

In the third stage, engineers are trained in institutions that teach the pragmatic engineering approach together with the acquired base of knowledge. The partnership developed between the faculties of engineering and science becomes synergistic. Each draws from its own specialty, while retaining the thrust that separates the two disciplines. As a result we can build bigger "pyramids," and we certainly have defined what a pyramid is.

If this model is general, and I believe that it is, it is reasonable to ask how the development of software engineering fits. Clearly it will be difficult to squeeze it into the 25 years of computer applications. In the early years programmers learned their skills through on-the-job training. At that time, there was disagreement about what constituted appropriate training for a programmer, with opinions ranging from mathematics at one extreme to musicology and romance languages at the other. One inevitable result of this confusion was the number of projects in which the result overran budget, exceeded schedule, and did not work as predicted.

The software crisis, officially identified as such in a 1968 NATO conference, resulted in a call for a new class of programmer who was to become the engineer of software development. The engineer would make certain that schedules and budgets were met and that the function promised would be doable. The engineer was to be the practical professional who would make certain that effort was expended efficiently and effectively.

In practice this idea became an attempt to form a discipline of training and perhaps even certification before the base of accepted knowledge was identified. Types of problems similar to those faced by the pyramid builders were given to the software engineer: Was there a sufficient body of practical and theoretical information that could become a curriculum in software engineering? Could the university faculty trained in software engineering teach a new software engineer what to do? Could the university faculty trained in software engineering do the experiments that determined the equivalent of what the ratio of height to base should be and then apply the results to other projects? Could the university faculty analyze the data accumulated, generalize the data by following the applicable scientific rules, and devise new experiments to further our knowledge of software engineering? Could all of this be done in a few short years?

These questions are but some of the challenges that software engineering has faced in its attempt to reach

a first level of growth. Curricula in software engineering did not exist because the scope of the subject had never been defined. No methodology was developed for experimentation. The models that were available were either from the natural sciences, where exact rules existed and the researcher was limited by technique and the accuracy of instruments, or from the behavioral sciences, in which the variability of the subjects caused a different level of uncertainty and a dependence on other statistical techniques. Software was a combination of both of these and required some level of cross-training to design and analyze the experiments.

The experiments that were done occurred on a small scale, and few of the researchers had the experience to scale up the results to a level that made sense when a million-line-of-code system was being designed. The measurements themselves were done in an atmosphere of uncertainty because we had not determined what should be measured and certainly had no standards for comparison. Results are rarely stated in terms showing the inherent accuracy contained in the data. Thus, it is not unusual to see documents that report results of estimates involving people and measurements of project attributes quoted at four or five significant digits.

All this leads to an era of great excitement, development, and confusion. Software engineering is experiencing a growth rate consistent with those of all emerging disciplines. Those of us who attempt to help it along must be ready for change in approach and change in direction, and, to paraphrase the story of the frog climbing up the inside of the well, be ready consistently to take two steps forward and fall back one.

What is software engineering?

The discussions regarding software engineering have all taken place without a formal definition that even a plurality of the concerned population has accepted. The domain of software engineering has been assumed to include three intrinsic areas of concern: software reliability, software management, and programmer productivity. One can usually find a body of practitioners who are convinced that one of these three areas is the most important and that the other two are secondary. Thus, a school of thought has grown up favoring software reliability and the drive to produce error-free code as the primary purpose of software engineering. The exponents of this approach have developed rigorous and formal languages to express specification and design. The goal of proof techniques, either constructive, following the school of Linger, Mills, and Witt,[1] or axiomatic, following Hoare,[2] is looked upon as the ultimate task of software engineering research and education.

A second school of thought has focused on the management aspects of software engineering. These practitioners have examined the planning and control techniques of large development projects in other engineering domains and have attempted to transfer these techniques to software development. The argument has been "If we could place human beings on the moon on a scheduled basis, we ought to be able to use these same techniques to write an operating system." The disciplined life-cycle approaches, which have been described in many places in the literature,[3-5] are examples of some of the end products of this group's efforts to date.

Each of these first two groups agree that productivity is a by-product of following their advice. The first group proves that the cost of error detection and correction is at least half of the development cost of any software component, and therefore the elimination of any errors first must increase productivity dramatically. Further, they argue, automatic tools will make certain that many of the error-prone activities of the past will be eliminated, and as a result, the new, more automatic approach will inherently be more productive.

The second group has a similar argument. If we know what we are doing, they say, and if we can plan and control more efficiently, there will be less backtracking because of misplaced and contradictory effort. The result of proper plans and controls will be to lower the error rate to an understood and controllable level while simultaneously increasing productivity. The automatic tools that we introduce to keep track of where we are will act as multipliers for the work of the software developers, making certain that we accumulate the data required to improve the process even more.

As a result, these two approaches have led the pragmatist to question the validity of the theoretician's experience and the theoretician to doubt the ability of the pragmatist to cope with the formal constructs that are viewed as necessary for error-free software. An example of the dissonance between these two groups has been the scant effort applied, until recently, to the theory of testing and to an understanding of how to deal with maintenance issues.

A further example of the problem is the scarcity of colleges and universities offering a degree in software engineering. No generally accepted curriculum has even been defined. The United States government is sponsoring a Software Engineering Institute[6] to address practical issues. It has been established at Carnegie-Mellon University, and one of the first orders of business of its education division is the creation of a task force or workshop to identify the core curriculum of such a degree.

For this paper, we use the working definition that stresses the practical nature of applying established principles so that efficient development occurs: the establishment and use of sound engineering principles (methods) to obtain economically software that is reliable and works on real machines.[7]

This definition stresses the engineering component of software engineering and takes for granted that principles do exist, have been discovered, or will be discovered shortly. It also assumes that the accumulated body of knowledge will be packaged coherently, so that in the not too distant future we will have degreed or certified software engineers.

The ideas that are embodied in this definition and that seem to be firmly in place are the following:

- Each step in the production of a piece of software should be designed to contain only independent, abstract components.
- Each step in the production of a piece of software must have a verification alongside it, preferably done by an independent agent.
- Each piece of work to be accomplished should be estimated for size and effort before it is begun and compared with that estimate on completion so that differences can be understood.

The development of software engineering. The first idea above leads to more robust designs and implementations. It allows for the isolation of function and data so that they can be constructed as cleanly as possible. It allows the designer to achieve intellectual control over a complex piece of software.

The second idea instills the discipline of reliability at all levels. Just as a carpenter should always measure one more time before cutting, so the developer should verify the correctness of the translation from the "plan" to the completed product at that stage. Similarly, the software engineer should validate at the earliest possible time that what *is* constructed is identical to what *should be* constructed.

The third idea is the engineer's way of developing new insight and rules of thumb. By approaching each part of the work process with this mental discipline, the software engineer is prepared for the early warning signals of possible problems. When completed, the analyses of both successful and unsuccessful estimates become grist for the mill of experience.

The software engineering evolution has followed several major streams.

This combination of three ideas is what distinguishes an engineering discipline from the *ad hoc* creations of both artists and artisans.

Software engineering: The major themes

The software engineering evolution has followed several major streams and has in the process gone through different eras in which the solution to the problem of developing software has changed dramatically. One stream that is visible as we look at the last two decades is the relationship between process and data. Ross and Schoman[8,9] observed during the early Structured Analysis and Design Technique (SADT) papers that data and process reveal one another's duality. Each represents the opposite side of the same coin. The nature of this duality stems from the observation that both must be present when an activity occurs. Examples in other areas are nouns and verbs, objects and operations, and passive and active voices. If we specify the process that occurs, we must also specify the data used by that process. This means that any analysis done to understand the contents of a system must include a data analysis component and a procedure analysis component.[10]

The apparent universality of the dual nature of data and process suggests that a fruitful approach to use in information processing might be to apply similar approaches in solving process problems and data problems. An example of this duality might be the

relationship between levels of abstraction in system design and data abstraction as drivers of system description and design.

The idea of levels of abstraction in system design states that the most secure system design will be done in a series of steps. Each step assumes the existence of an architecture that represents a complete description of the system required to complete the step now being done. There may be many strategies for defining the appropriate layers, with one approach defining function successively nearer the exercise of the actual hardware[11] and another perhaps segmenting a design into independent components, or vocabulary approximating the vocabulary of the application.[12] The strategy employed prohibits any one layer from communicating with anything other than the layer directly beneath it in the ladder of abstractions. This approach isolates function so that unexpected complex interactions are held to a minimum.

When the concept of abstract data items was introduced,[13] some of the thinking was very much the same as in function abstraction. The intent was to create data structures and keep them isolated from any other component of the system. No inadvertent access was to be allowed, and no unexpected complex interaction would occur. A package was to be created as a combination of the data storage access routines and all of the functions that would be performed on a particular data structure. All users would have to be funneled through the code supporting the abstract data item.

The thinking that went into both of these concepts is similar and is an example of the dual approaches one can take toward both data and process. A further example comes from a comparison of two conceptually similar papers, one by Bohm and Jacopini[14] on the number of constructs required for a general program, and the other by Mills and Linger[15] on how to create a program with a constrained number of data structures.

In a similar fashion, consider the relationship between data base systems and program design libraries. This dual relationship appears quite general, and one should expect that every improvement in one of the two will eventually result in a symmetric improvement in the other.

A second stream of the software engineering evolution is the scope of the development process that must be used to manage a software development project. The view in the 1950s included only the writing of code. The view in the 1960s extended toward design on one side and test on the other. By today it is recognized that a development cycle must include all aspects that go into determining the contents of a software product prior to implementation and must be extended to consider what may take place in the future when the system will no longer be used. This increase in the scope of a development cycle has extended the purview of system analysis into the world of business strategic planning for the application community and into product strategy for the system development community. Along with this extension has come the realization that the time span now included is so long that the attributes of the delivered products no longer match the requirements of the consumers of the product. The rate of change of user concerns is faster than the rate of production of software to support these same users. This situation leads to the inescapable conclusion that we are faced with a paradox. How do we include all of what is needed for an efficient development cycle in a time span short enough to satisfy our customers?

Software engineering can be said to have begun in the late 1960s.

Throughout the years when software engineering was developing, many individuals made enormous contributions and advanced our understanding of what needed to be done. Not all of the advances were appreciated in their own time, and some were forgotten completely. Any attempt to provide even a rough chronology will only look at highlights and try to plot the crest of acceptance. For example, ideas that may first have been presented in 1970 but did not achieve broad acceptance until the late 1970s will be listed, with only minor historical license, at the time of their acceptance.

The eras of software engineering are roughly these: We first looked for a single magic solution, with structured programming as the promise. When this was insufficient, the second approach suggested that

the use of a consistent methodology would solve the problem. This approach did not solve the problem either. The third solution suggested that, in addition to a consistent methodology, one ought to create an environment to support the developer. This combination, it was believed, would create the best programming environment. Of course, there are no perfect solutions. Today some are suggesting that expert systems will help. The best solution will be the expert software engineer, assisted by a large variety of tools, using experience, judgment, and creativity to advance the state of the art. This is just as it is in all other engineering disciplines.

The first era. The development of software engineering can be said to have begun in the late 1960s with the observation by Dijkstra[16] that "Goto's are considered harmful." This observation began the first really systematic investigation of how the structure and language of a program affected the way it was produced and the way it behaved. The structured programming revolution is the first of the waves of change suggested to the programming community to produce a product that was more reliable, was produced more productively, and was more manageable.

If you belonged to the school of thought that stressed the reliability of software products, you viewed the direction as consisting of the following theoretical argument. Any logical construct could be built out of no more than two or three program structures, sequence, iteration, and choice. These items were equivalent to the programming constructs of next statement, DO-WHILE or DO-UNTIL, and IF-THEN-ELSE. The unrestricted branch statements were excluded (either totally if you were a purist or until absolutely required if you considered yourself to be more pragmatic) as causing problems. Procedures were to be turned into proper programs with a single entry and a single exit point so that a minimally constructed algorithm, small and compact, easily verified algorithmically as being correct by a manual or automatic proof technique, could be produced. It was only a matter of time, you believed, until efficient theorem-proving algorithms would be made available to the development community. At that time all errors would be gone.

If you were a pragmatist, you stressed the human factors of eliminating goto statements and keeping the program constructed from a minimal number of constructs. This group added the idea of variable names which were easily understood as being a way of decreasing the error-proneness of the software. *The Psychology of Computer Programming* by Weinberg[17] was a hallmark publication in this era.

In this work, Weinberg put forth the idea of egoless programming as a way of increasing the reliability of software. The notion was that many of the problems introduced into software arise because of the defensiveness that the programmer feels about the program just written. Programmers see what they expect to see, and do not see those "errors" that will make them appear to be less than perfect. The solution suggested in the book is code reading. Each programmer should be self-motivated to share source code and accept criticism. In this manner many perspectives are used to create an error-free system.

Technology transfer is a slow process.

Code reading inevitably led to walkthroughs, design reviews, and inspections as being ways to detect errors earlier because it was less expensive to produce systems this way.

The idea of a development life cycle had already taken hold, although the important part of the life cycle was considered to be coding. The early stages of requirements were important but not crucial; the late stages of maintenance were only an unavoidable cost of using software. When the process of producing correct programs had finally been accomplished, maintenance would simply fade away.

This era, like the ones that followed, never formally ended. There are still individuals who believe that the introduction of a few simple ideas will solve all of the problems. Technology transfer is a slow process, even in a field as fast-moving as information technology. A recent study[18] suggested that the length of time required to transfer an idea into the culture of the recipients is 15 to 20 years. At that rate, if structured programming had been introduced in 1968, it would have been widely accepted by 1985.

This acceptance rate does not mean that everyone will use structured programming (this is just an example), only that no one will disagree that it should be used.

In those locations where structured programming and inspections were instituted, management did see an improvement. The general rate was three to seven percent[19,20] better than before these techniques had been introduced. The feeling was expressed that the development cycle needed to be expanded; the approaches should be applied not only to coding but to design. The portion of the development cycle that needed to be managed was extended to include design on one side and testing on the other. Structured programming was supplanted by stepwise refinement and structured analysis, to name just two approaches.

The dual principle was expressed in the relationship between strength and coupling in the works of Yourdon and Constantine[21] and Myers.[22] Strength was related to the purpose of the piece of a program being worked on. The more cleanly the purpose could be defined, the greater was the strength of the component. The ideal was to be able to describe the purpose in a simple sentence which contained no connectives—no ands, buts, thens, or whiles.

The notion of coupling extended these ideas to the realm of data, data sharing in particular. It was held that any pieces of data or code generally available to many programmers were error-prone and should be eliminated. The revered FORTRAN common statement was now on the list of ten worst enemies of program reliability.

The two ideas of strength and coupling, together with the previously cited work of Parnas[13] describing what criteria should be used to break systems into smaller pieces, firmly established what a module should be. The clarification obtained by separating strength and coupling began the inevitable path toward acceptance of data abstraction as a concept equal to Dijkstra's levels of abstraction on the procedural or functional side.

The second era. The era of methodologies was driven by the idea that the developer needed to have a general meta-approach to specify what should be done next. This idea would make certain that no matter what the level of understanding, the next thing to do would be known. The concept was exemplified by general rules that were laid down, each tending to consist of multistep processes. An example of the kind of instructions written down as methodologies would be this excerpt from the Jackson Design Technique:[23]

1. Consider the problem environment and record our understanding of it by defining structures for the data to be processed.
2. Form a program structure based on the data structures.
3. Define the task to be performed in terms of the elementary operations available, and allocate each of those operations to suitable components of the program structure.

The methodology expressed by Jackson is very different from that expressed in Linger, Mills, and Witt[2] and in the works of Yourdon and Constantine[21] and Myers.[22] The breadth of approaches made popular in the era of methodologies extended across the dualism of data and process, with the stepwise refinement of Wirth[24] and Linger, Mills, and Witt[2] on one end. The middle ground was held by structured analysis, as defined by Constantine and his followers. The data side of the spectrum was espoused by both Jackson[23] and Warnier.[25]

The user was at a loss to decide which of these approaches made the most sense. Which one was right? If there was not one that was more correct than the others, the question was changed: Which one is right for me at my location? There were no definitive answers to these questions. In his review article, Bergland[26] placed these approaches in perspective and highlighted reasons why none were perfect. A recent review article by Yau and Tsai[27] gives a 1986 perspective on comparisons.

Software engineering has not yet measured the differences between approaches, and very little seems to have been done to establish an experimental difference between various methodologies. The closest to an early definitive measurement was provided by Basili and Reiter,[28] who showed that any disciplined approach, consistently applied, is better than lack of discipline.

A second dimension was added to the discussion of methodologies as those people who were mathematically oriented used formal language structures to express their ideas. The contention was that the ambiguity of natural language caused confusion in system requirements and specification. This ambiguity was interpreted differently by members of a

development team and inevitably led to confusion, inconsistency, and contradiction. The problem would be solved when the language of expression was exact. The formal structures introduced seemed directed at the theoreticians to the exclusion of the practitioners.

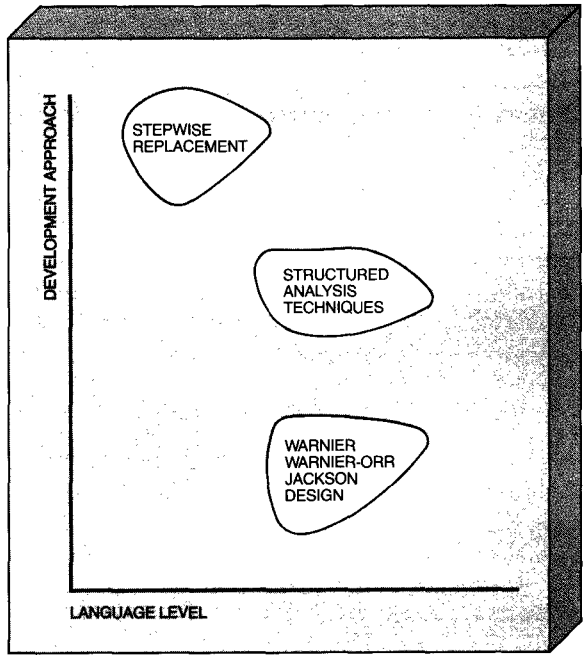
In their examination of the application development environment, Gillenson and Goldberg[10] laid out the combinations of approaches together with the formality of the language on a two-dimensional plot to show the diversity of methodologies and compare them with one another. This chart is reproduced in Figure 1.

For these structures to become widespread, there would have to be a broad educational effort designed to change the way of thinking of a large group of people. If the period of time for technology transfer suggested previously in the discussion on structured programming is valid, any such education program would require something like 15 to 20 years to be effective, which in this context would mean "accepted as the right way to do things by a majority of the affected community." A good starting date for an education process might be October 1977,[29] when the IBM Federal Systems Division began its celebrated education program, later transferred to the rest of the IBM commercial programming population.[30,31] General acceptance would then exist in 1994 (again, not when everyone would use a formal language, but when all would agree that one should be used). This effort provides an example of why progress appears slow even in an accelerated environment.

As with the first era of change, software management ideas were introduced on an entirely new level. If during the first era most thought of the life cycle in a constrained fashion of design, code, and unit test, those in the latter era looked at system analysis and design and system testing as crucial elements of the development process. Compare Aron's approach in the two parts of *The Program Development Process*, Part I[32] having a 1974 date and Part II[33] a 1983 date.

The life-cycle concept also began to assume some of the attributes of a set of discrete steps. The attempt by Fagan[34] to introduce the ideas of quality control via the inspection process used the notion of discrete tasks having entry and exit criteria to delimit them. By implication, this method provided a series of stages that could be independently verified before proceeding with the next stage. The method also provided identifiable tasks during which pieces of work having a tangible end product were produced.

Figure 1 Relationship among development methodologies[10]



The product could be estimated before work began and measured after it had been completed. Although all of this was not seen at the introduction of inspections, the success of the technique in improving defect detection led to widespread acceptance and would allow the introduction of more advanced management concepts in the next era.

The extension of life-cycle management and concern into the testing domain began to focus attention on software maintenance. It no longer appeared to be inevitable that maintenance as an activity would cease to exist. The studies[35] began to suggest that there were three separate related activities being charted. The first, of course, dealt with correction caused by human oversight or error. This activity could be reduced with better attention paid to error-detection mechanisms.

The second cause of maintenance was due to the natural evolution of the environment in which the program resided. This evolution was interpreted as meaning natural growth in business volumes, new classes of function wanted by users of systems, new devices that needed to be supported, and many other changes that were made by choice because of justified business decisions.

The third class of maintenance occurred because of mandated change. When the United States government changes the Social Security laws, each payroll program must be changed. When a company decides that all application programs will use the same data base system, each application program must be modified. Not all owners of these programs would have voluntarily chosen to make the change, even though good global business reasons exist for attempting such changes.

The role of maintenance was given some philosophical support in the work of Putnam.[36,37] He studied the data file at the Rome Air Force Development Center and came to some conclusions based on the application of Norden's work[37] on Rayleigh models for development processes. The originally astonishing conclusion he drew was that if (1) a Rayleigh curve describes software development, and (2) the time during which the largest number of people are employed developing the system to be delivered is just at or around customer delivery, this inevitable conclusion follows: 39 percent of the cost is in development, 61 percent in maintenance.

Although there is no universal agreement that an ideal development cycle will follow the Putnam-Norden model, there is agreement that some split in development–maintenance costs such as 50–50 or 40–60 is to be expected. As a result of this more mature view of maintenance, continuous modification was understood to be inevitable, and software engineering needed to address the designing of generations of programs rather than a single version at a time. This view would cause a further strengthening of the need to make program design and coding simpler. It was not enough to make it work; it also had to be made easier to modify. Because of such issues, the concept of configuration management and control began to be looked at as having potential utility in the production of larger pieces of software. The successes of the early space program served as a model of project management for software development, and these concepts are an example of the ideas gleaned from the program.

There had been many earlier attempts to transfer the project management concepts used in the space program into software development. The network scheduling techniques PERT (Project Evaluation and Review Technique) and CPM (Critical Path Methods)[5] use directed graphs to connect the identified tasks of a project and place them in relationship to one another. The combination of predecessor, successor, and duration, identified for the piece of work to be done, creates a visible model of the flow of the project and identifies the critical path of subtasks that determines the rate of completion.

Configuration management and control is the discipline that organizes, documents, and tracks any system consisting of multiple parts. It is charged with identifying the components and their relationship to one another, controlling the changes made to a configuration, keeping track of the status of each element of a configuration, and providing the mechanism for satisfying an audit of the compliance of a system to its required configuration.[38] The application of this discipline to software development has been driven by the U.S. Department of Defense, as evidenced by the 1968 Military Standard.[39] There also exists a more recent IEEE standard for Software Configuration Management Plans.[40]

A new addition to software management and control began with the introduction of programming dynamics.

The history of the evolution of Software Configuration Management is given in Bazelmans,[38] and additional information is contained in the IEEE tutorial on the subject.[41] There were isolated pockets where each was found useful; however, there was not enough commonality and discipline in the development process to allow for sharing of work. Each installation had to begin anew and rediscover areas of applicability. This lack of sharing, together with a programming tradition of free-form work styles, made the use of these project management techniques less efficient than their designers had hoped. But the seed was planted.

A new addition to the repertoire of software management and control began with the introduction of programming dynamics by Belady and Lehman.[42,43] From 1969 through 1980, they published papers

indicating that large systems seemed to obey the *laws* of software evolution. The analogy was drawn from statistical mechanics and thermodynamics. Systems were not able to grow indiscriminately in size without suffering a form of *arthritis*. Constant additions to already existing pieces of code scrambled the original design integrity, causing less efficient, more error-prone code. This phenomenological approach to measurement has worked well in other areas of science as a precursor to a more axiomatic understanding of the underlying science. Here, too, we have the beginnings of a more exact science.

The toolkit that the software professional used at work was still very personal. Each tool was chosen to satisfy a particular need, not to provide a way to move the work from life-cycle stage to life-cycle stage. Even among one development team, professionals chose to use or not to use particular tools. Of course, as development became more complex, the magnitude of the task grew, and more tools filled the toolbox, covering ever more of the development process.

Some particularly successful examples began to be widely used, and families of tools became commonplace. Almost none of the tools were designed to support a particular development methodology. Each of the popular methodologies remained a mental discipline that answered the question: What shall I do next? None of the methodologies were embedded in an automated factory for the development of software, even though it was now understood that this had to be the next step. The use of a particular methodology did not bring the quantum increase in capability that some expected; productivity increased just seven percent per year.

No discussion of this period would be complete without describing the introduction of Ada,® the programming language which was to be the embodiment of all that software engineering had learned up until that point in time. One of the largest users of software is the U.S. Department of Defense (DoD). It was using virtually every language and machine architecture that had been created. The software ranged in complexity from simple programs to complex communication, command, and control programs to be embedded in hardware components.

In 1975 a search began for a better language that could be used in all DoD applications. This search culminated in a language competition in which a language design that was to become Ada was chosen. Toward the end of the competition a similar attempt was begun to define the Ada Programming Support Environment which identified the tools support to go along with Ada.[44] Booch[45] contains a historical overview of the timetable and events that led to the current description of Ada. As of today, the language has been defined, it has been described in both technical material and popular texts, and the initial compilers have been written. Full production compilers and large systems will follow soon.

The Department of Defense instituted several initiatives in information processing. The two already referenced are the definition of Ada and the creation of the Software Engineering Institute at Carnegie-Mellon University. A third initiative is the Software Technology for Adaptable, Reliable Systems (STARS)[46] program. STARS led to the charter of the Software Engineering Institute. STARS is seeking to create an integrated, automated environment to cover all of the development life cycle with a management and measurement system that brings the newest technology to bear as quickly as possible. This environment involves measurement, experimentation, tools, and education.

The third era. What then should be done to increase our ability to produce correct software? All estimates of the pent-up demand pointed to an environment in which the major limitation on new software applications was the insufficient supply of programmers. The limitation stemmed from the inability of the software community to produce the quantity of software at a level of complexity specified by the requirements. The solution appeared to be automation. Rather than have manual methodologies supported by a set of unrelated tools, developers began to talk about software environments containing integrated tool sets to take development from requirements into design, code, and test. The tools would accumulate the management statistics required and would become the catalog of component parts. The catalog of component parts represents the bill of materials for a software product of manufacturing. For the first time software development would consist of a traceable product. Each requirement could be connected to all realizations of it in code, and because of this string connecting dependent pieces, we would be able to make certain that the inevitable changes did not cause unforeseen side effects in other pieces of a system.

Software configuration control and management have been achieving a wider range of acceptance and

have begun to enter the world of the programmer. In this third era, software as a potentially profitable product has excited the imagination of managers and entrepreneurs alike. Together with the potential profits have come the product project management techniques found necessary in other manufacturing environments. The discipline of using a development methodology has led to the discipline of project control.

The use of environments forced the move toward more formal requirements and design languages.

The technology of development had accepted the ideas of data and procedure abstraction. The use of environments forced the move toward more formal requirements and design languages because they needed to be machinable. It is but a small intellectual step from insisting that all aspects of a design be captured to insisting that it also must become executable. This requirement had strong appeal because in the process the machine–programmer interface would be pulled higher, and part of the productivity and reliability improvement would come about from eliminating much of the coding. Coding had already been limited to 10 to 20 percent[47] of the invested effort in software development.

The Ada-based languages now evolving allow for better verification of the design and the code that flows from it. They are more formal, and hence have fewer (if any) unresolved ambiguities that lead to design errors in a completed product. Ada introduces the concept of a Package[45] as a set of computational resources pulled under a single boundary. Previously, abstractions of data and procedure were used as entities whose purpose was to isolate the developer from the machine, thereby diminishing the potential for error. The package concept takes this idea one step further by allowing the designer to choose the inherent elementary ideas in the physical aspects of the real problems to be solved. Thus, problems involved in designing a system to implement on-line circulation control in a library would be able to identify packages representing nouns such as collection and verbs such as borrow, reserve, and return. The librarian would then be able to assume a more productive role in creating and validating the software application.

The resulting design should read like the subject specialist's description of the scope of work. The implementation specialist designs the packages and produces the connecting logic that ties together the data flow aspects of the real-world circumstance. It is one step closer to capturing design in executable form and one step closer to solving the problem of requirements that change faster than our ability to produce the code to implement them.

The defect-detection methodologies include a patterned set of inspections which fit into those cracks between distinct stages of a development process. We now understand how important these methodologies are and expend the effort to detect errors as early as possible. In a graph produced by Boehm,[5] it was shown that the ratio of the relative cost to fix a problem decreases when it is found early in the development cycle. Testing itself has been subjected to theoretical analysis, and more attention is being paid to test tools that assist the developer in producing an error-free product. Zero defects for all products is something like absolute zero in thermodynamics—you may never reach it, but you can come very close in approaching it.

The strategy has moved away from defect detection to defect prevention. Methods of design verification, which the individual developer can use, make it possible to "guarantee" that the design and its code are exact translations of the specifications defining what is to be done. This verification is made possible by the flow of ideas in similar if not identical languages from specification to executable code. The support environment has begun to be defined.

The next step in the completion of the support environment occurs when the tools supporting the use of the design languages meld into the tools supporting the remaining portion of the development process. The shift to a defect-prevention strategy will eliminate the insertion of errors into a product. Testing is required to find them and take them out. However, errors of omission, which occur because a software product either is performing an inappropriate function or has neglected to include an operation that the end user really wants, have still not been eliminated.

Errors of omission are caused by many factors, including inadequate gathering of requirements at the beginning of software development and problems with the consistency, completeness, and correctness of requirements for several of the independent pieces. Unfortunately, the length of time needed to produce a software product is long, and many of the symptoms of errors of omission do not appear until the product is completed.

Requirement definition is a problem that appears on lists of problems to be solved, from the first[48] to the most currently[49] compiled. The early software engineering focus viewed the problem as requiring a more formal language of expression. The use of a natural language led to ambiguity, incompleteness, inconsistency, and contradiction. The grudging use of approaches such as the Problem Statement Language/Problem Statement Analyzer (PSL/PSA)[50] improved the situation somewhat but not for everyone nor for every instance of its use.

Similar approaches derived from PSL/PSA use Entity-Attribute-Relationship models to name the objects whose properties are being modeled in a software product and to show how these objects relate to one another. These objects are described in terms of their attributes and how they relate to other objects similarly described. Once the tables of information have been entered in machine-readable form, a network of relationships can be constructed and used to determine definitions in which descriptions do not match. This network becomes an early tool for picking out ambiguities and inconsistencies.

Another approach that becomes appropriate after the initial set of requirements has been analyzed and the "fuzzy cloud" of raw user requirements cleared up is the finite-state machine.[51] The model used to describe a system whose implementation is proposed is structured as a series of states and events. An event, for example, the occurrence of a particular piece of data, causes a transition from the current state to some other state. If the high-level designer can assign identified functional requirements to a state with defined transition rules, it should be possible to model the logical completeness of the set of requirements for which the proposed system design has been suggested.

Each technique is capable of providing some automatic assistance in improving requirements but is still unable to solve the complete problem of providing customers with a product to satisfy their needs *at the time they receive the software product*. Their requirements depend on an understanding of what they are to accomplish. This last item changes the first time the enhanced environment (hopefully) became available for use; i.e., it is impossible to hit a moving target.

One view of a software development process is as a large "V," shown in Figure 2. In this view, the top left leg of the "V" represents the earliest stages of requirements, and the top right point of the "V" represents customer acceptance. As you draw horizontal lines across the "V," the right leg validates the work done on the left leg. Therefore, the earliest validation of requirements occurs at the time the customer accepts the package. For large systems this time is many years after the statement of needs was compiled. The solution to this impossible dilemma, suggested initially by Brooks[52] and becoming generally accepted in the third era, is rapid prototyping.

The idea is to build a working model, simulation, interpretation, breadboard, etc., of the final product and to allow the user to "play" with the prototype and suggest improvements to the concept. The final prototype becomes the specification that enters the design methodology, to be implemented in an efficient and cost-effective manner.

There were early difficulties with accepting prototyping as a step in software development because prototyping foundered on the attitude that a throwaway failure was being produced. The developers could almost be seen thinking, "If it is really bad, the work is not worth anything; if it is any good, make it better and ship it."

A more appropriate view is to treat prototyping as the implementation portion of an independent development cycle, the final output being a specification from which the operational system will be built. The "V" described above is replaced by a lopsided "W" in which three distinct operations might occur: First, analysis of raw requirements leading to a statement of the contents of the system; second, a finite-state description, which may even be executable, to test the logical completeness of the proposed elements; and finally a set of connected screen designs which can be tried out and, with a user's assistance, can lead to a very early evaluation of the system. The result is an interface specification to build the working product. This view is shown in Figure 3.

A management advantage gained by following a scenario of this type is to define and be able to track each of the requirements that must be satisfied in

Figure 2 The "V" of software development

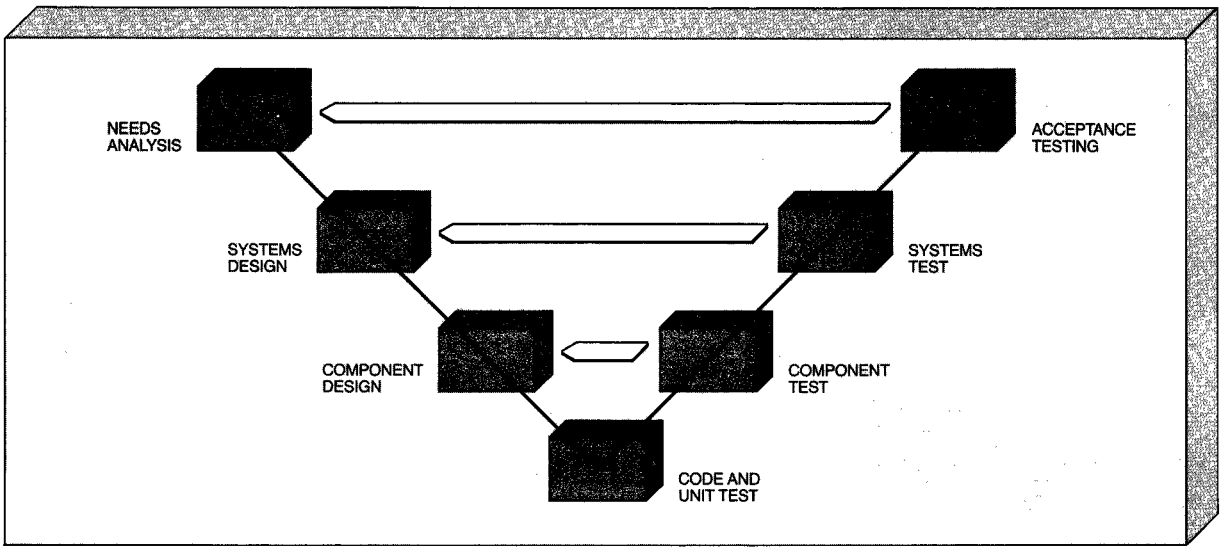
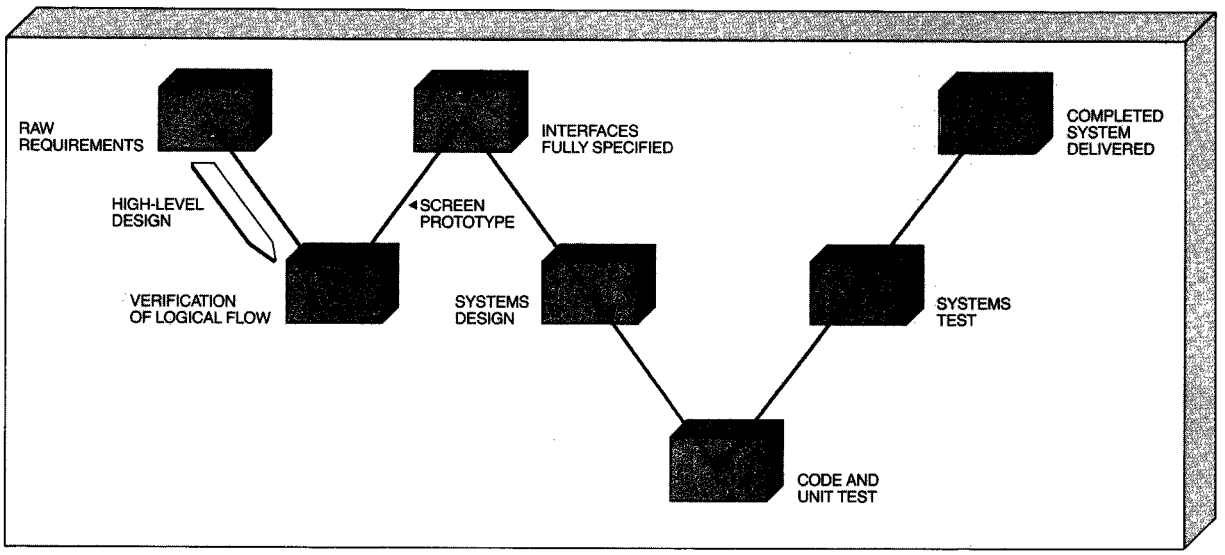


Figure 3 The "W" of software development



the final product. The use of machinable descriptions to enforce consistency, completeness, and correctness produces a version of the requirements that can be entered into a configuration management data base. The accepted prototype, intended to become the specification, defines a baseline as a marker against which all changes can be measured and logged. The combination should provide a mechanism for tracking each function through a product, such as in the bill of materials processor and the diagrams of parts explosions of manufacturing systems.

What evolves in this era of development environments is a set of integrated tools allowing the designer to be free for the creative efforts of design. The bureaucratic paperwork of keeping track of details is best left to the machine. The time-consuming effort of making certain that we have not accidentally changed a name or an attribute is done automatically. The questions that the bill of materials was invented to answer (i.e., Where is each named part used? What are the components of each subassembly?) can now be extended into the domain of software construction. This scheme begins to provide a consistent, integrated library package for parts inventory and version control.

To complete this dream will require an incredible amount of machine power and machine availability. The solution appears to be to provide each developer with a personal computer workbench acting alone or in concert with a host or a network of peers throughout the world. This solution creates its own complex software design requirement, in which rules of distributed processing and software architecture are tied to management control issues.

We now come to the change in management style toward which the third era is headed. One name for this approach is software process control management.[53] It borrows its ideas from other process control environments and applies a quantitative management technique to the software development process. It attempts to change the thinking patterns of the software development manager by creating an atmosphere in which each step, large or small, is preceded by a numerical estimate of what is to be produced with respect to size, cost, quality level, time, etc. The manager is directed to end each step with a measurement of what was achieved and a pattern of thought that asks the questions: Am I on target? Why? What's wrong? What does this mean about other estimates I have made? Will I make my goals? The result is better control and incremental improvement activities. This approach is an example of measurement-directed software management.[54,55]

The approach outlined above is a natural outgrowth of the inspection strategy introduced by Fagan[34] many (in software engineering chronology) years ago. It depends on the "crisp entry and exit criteria" that delineate stages and tangible pieces of development. In the "process control management" terminology, the entry and exit criteria referred to above have been translated into the "ETVX" (Entry-Task-Validation-Exit) paradigm of the IBM programming process architecture.[56]

Figure 4 The first three ages of software engineering

ERA	TECHNOLOGY FOCUS	MANAGEMENT FOCUS
FIRST 1960s	TECHNIQUE EXAMPLE: STRUCTURED PROGRAMMING	INDIVIDUAL
SECOND 1970s	METHODOLOGY – STEPWISE REFINEMENT – STRUCTURED ANALYSIS	LIFE-CYCLE COMPONENTS
THIRD 1980s	ENVIRONMENT	PROCESS

As a further consequence of the increased use of metrics to control a development process, a way to have a true software engineering laboratory is finally provided. For the first time, the manager of software development is given the incentive to participate in the data collection activity and become an active participant in advancing software engineering knowledge.

Figure 4 summarizes the three eras through which software engineering has progressed.

The future

We are still investigating the advantages to be gained by creating workbenches and development environments. Much of the information to be gained about the advantages of an integrated tool structure will come from Ada environments. Certainly we will have to continue the steady pace of development if we are to improve our engineering discipline. The technology that is being developed will draw out the utmost from the development processes now in place. To the extent that we can stabilize these processes for some length of time, we will be able to learn much more about the way development occurs. Consistent improvements come about only through the steady accumulation of data. Our experience with the industrial development of software provides no examples of spectacular improvements across the board. No single idea has burst forth to change the nature of development. It has been a steady improve-

ment year after year. This pace can only continue with a disciplined approach and a steady accumulation of knowledge.

The reason for the lack of sudden improvements is not known. It may be as simple as having to wait the time needed to transfer the technology we have developed to a large enough segment of the population so that a large impact can be noticed. Another possibility is that we may have simply missed the key ideas leading to the breakthroughs for spectacular improvements. The third possibility is that there is no hidden approach, and we will continue to make progress at a slow but steady pace.

Regardless of whatever combination is closest to reality, we must assume that hard, steady work is the only sure technique for improving productivity and reliability, and we must begin with our accumulated base of knowledge. If this is true, the future direction must fall within these broad categories. We can improve the way our mechanisms operate, deriving in this way the final measure of improvement possible. We can change the way our mechanisms operate by eliminating steps or drastically redesigning the sequence of steps to accomplish the same work with many fewer resources. This change would mean a reformulation of the life cycles as we know them today. Finally, we can eliminate the steps associated with producing software entirely by having the end users describe their problems and using those descriptions to produce the information and systems required.

The first of these three approaches is the domain of the engineer. Experience teaches us where the gears need oiling and where the belts need tightening. The software engineer examines the development process and determines where the critical bottlenecks reside. These bottlenecks are eliminated one at a time. As each layer is peeled back, the process becomes more efficient, and the next bottleneck is revealed. Such slow and steady progress causes productivity improvements on the order of three to five percent per year.

The thrust today to study, gather data, analyze, and streamline development processes is an example of the approach of applied software engineering. It is a necessary task for all programming organizations and is a prerequisite for real progress. However, it only sets the stage for the work necessary to get the next level of improvement after the diminishing returns of the tuning process. There are limits to the total productivity to be gained in this way. Radice[47] estimates that the maximum improvement possible with today's development process is four to one, because software is a labor-intensive occupation. The approach then must be to redesign our basic way of thinking about the way software is developed.

If other engineering disciplines are assumed as models, a sensible approach is to identify or develop software building blocks that can be reused in multiple applications. Just as interchangeable parts allowed the industrial revolution to reach the average consumer with affordable products, so too the use of interchangeable software parts may make it possible to decrease the labor-intensive component of software development.

This approach is being examined seriously. A recent issue of the *IEEE Transactions on Software Engineering* was devoted to this topic.[57] One report from Toshiba[58] described a real-time process control "software factory" in which customers accepted the equivalent of four million lines of assembler code, of which 50 percent was reused code. Other reports circulating through the software community promise or hint at reuse rates of 85 percent.

If there is such great potential for increasing productivity in the consistent reuse of code, why have we not used the already completed billions of lines to ease our workloads? There have been at least three inhibitors to the widespread reuse of existing code. The first results from the multiple languages that have been popular. Each is unique in some way and cannot coexist with programs written in some other language without a level of effort equivalent to writing the piece over again.

A second inhibitor stems from the dependence of a piece of software on the environment in which it is embedded. A design philosophy formerly used, to what appeared to be great advantage, was to share data among the modules and subroutines of an application. It was felt that the programming would be easier and the performance would be better because it would not be necessary to name every data item. A consequence of this technique was to require the description of the data environment in which the software fragment would be found, in order for it to be recreated for reuse. This description was possible only in a few instances, and consequently the work produced could not be shared. The introduction of abstraction as a key element of software design changes the language and environment restrictions that made reuse so difficult, and specific pieces can be commissioned to act as building blocks in specialized environments.

The next problem is how to determine when a component has already been written. The term "information explosion" referred to the quantity of printed material produced and available in our information-rich culture. Information retrieval experts began to design and create software products capable of assisting the browser in locating and retrieving potentially relevant material.

Similarly, the number of modules being written and stored in a haphazard fashion is also exploding. For the written word, a centuries-old cataloging tradition exists. Useful schemes and descriptors have been written, and we are trained in their use in elementary school. However, the module description language and its associated descriptors are not commonly available, and an implementer must serendipitously come across a used module that can be inserted into a current project. The Ada construct of packaging should increase the potential for reuse and lead to an improvement in software development methods.

Code is less portable than design.

A further application of this approach is to consider the reuse of design fragments.[59] Code is less portable than design and has by now become the least expensive segment of the development process. The basic design of an algorithm or procedure can be transported across architecture boundaries. For example, sorting algorithms are rarely topics of crucial importance. We have understood the relative merits of each of the algorithms for some time and know how to choose which will work best for a current need. Many more examples of this class exist, and as specification languages become formal and machinable, we will begin to accumulate libraries of design fragments that can be retrieved and translated into the architecture of the machine we are working on.

An additional benefit of reusable software will be the increased reliability of the systems produced. It will be possible to choose components whose defect histories are well known and which have been shown to behave in a reliable manner. Ringland[60] reported that the reliability of a software product was improved as a consequence of software reuse.

The second alternative of modifying the process and thus eliminating steps is also being pursued with vigor. The ideas behind this approach include most of the fourth-generation language techniques[61] that have been fruitful in the recent past. The approach is that of the "almost complete program." The designers of systems produce general-purpose application generators which are completed by the individual who has a data processing program to produce. The standard pieces of code required to access the data base or the input/output devices are already written. The effect of this combination is to produce an environment in which only the details of the problem at hand need to be entered into the application generator. Most of the extraneous design issues, those related to the details of the hardware and systems in which the problem is embedded, are resolved for the designer, thus improving the achieved productivity by a considerable amount. Included in this broad classification are report generators, query processors, and screen design aids.

This approach has been most successful in applications programming, especially for those applications in which the efficiency of execution is least crucial. In system production, however, there have yet to be sufficient inroads to make this approach useful. It does point to a direction in which an application support environment may be able to provide some more of the common functions and free the software engineer from the need to invent similar repeated components over again. Notice that this is a special instance of the principle of reuse being applied.

Another example of modifying the development process is the use of software prototyping. As described above, prototyping would allow the developers to get an earlier warning as to the areas of concern in the programming system as they see it. Additionally, prototyping could be used as a participative design aid. The developers would work with the ultimate users, who would suggest changes to the initial version. Each change would bring the final form closer to the needs of the ones for whom the system was created.[62] This approach includes systems testing as a part of the requirements process, thus tying what was the first step to what was the last step. In those cases where it is applicable, this approach has the potential for improving the speed with which programs are turned around.

The third alternative, doing away with the process, must also be explored. The fourth-generation languages move in this direction. The use of expert systems to diagnose what a user requires is an additional step in this direction. Application knowledge in the form of standards, the layout of forms, flows of information, job descriptions, and organization structure could be matched to catalogs of potentially reusable design fragments, dictionaries of data definitions already captured, and modules and programs already written. The software engineer would direct a process in which pattern matching would occur under a set of rules or axioms that describe what is wanted, thus slowly approaching a system design. This method is a guided process, with intuition and experience supplied by the software engineer and high-speed pattern matching supplied by the computer.

Our understanding of what is to be measured has matured.

Although this method cannot be carried out today, some applications of knowledge engineering and artificial intelligence have begun.[63,64]

The alternative approaches to improvement assume that we can make intelligent choices about future directions. The major themes of technology insertion and management control assume that we have a sufficient amount of information to choose the direction in which we wish to head. The transition to an engineering discipline begins when information to analyze is available. This transition has finally begun to take place. Software development processes are beginning to stabilize, and the software engineers have accepted the concepts of goal-directed management. It is now possible to identify stages of development at which specific pieces of work will have been completed and compare the goal that management set at the beginning of development with what has actually been achieved. Comparisons with previous instances of using this process help set the goals, and stability makes the comparison meaningful.

Our understanding of what is to be measured has matured. It is possible to differentiate between those measures focusing primarily on a software component, those focusing primarily on a producer of software, and those which measure a software process. The measures that concentrate upon the software component deal with issues of how efficiently it runs, how much space it consumes, the number of errors it still contains (none after shipment), etc. The measures dealing with the role of the producer are concerned with the productivity and work quality of that individual. For example, programmer variability is well known to be of the order of 10:1.

Measures of software process are concerned with how well a set of steps allows the development team (or single individual) to produce a final product. The measure is of the efficiency in converting resources into a finished product. Examples of this might be the effectiveness of Inspection Step A in finding errors and the ability of Tool B to analyze requirements assertions for correctness. By separating measures into various categories of applicability, the software engineer can distinguish between the ability of a development process to assist in producing software and the effect of a particular technique. This ability is still rudimentary, but conceptually it is significant.

Making comparisons between diverse environments is still impossible. However, the comparison can be made within local, well-understood environments, the same location, application type, or machine architecture. We have not yet defined a set of universal measurements that can be compared across diverse borders. However, we *have* begun to experiment,[65] and this will eventually lead to a better understanding of measures and their use.

The power of the personal computer is leading to an increased use of tools at the programmer's desk. The programmer workbenches are providing the raw processing power for each individual to do editing and checking that formerly required large machines. When the history of the development of data processing is written some decades from now, the key tool that will be pointed to as improving the lot of the individual developer will be the editor and text processor. This tool has opened the way to record information about programs quickly and to manipulate the programs directly. This, in turn, has led to the on-line syntax checkers, debuggers, etc. making up the programmers' workbenches that will support future environments.

Finally, the attention being paid to validation and verification of the final product must be stressed. Whether it is the drive begun with structured programming concepts in the 1960s, or the management stress on productivity and control, the central point is that the most productive environment exists when no rework is required and we get it right the first time. This concept unites all of the elements of software production in a single theme and is the prerequisite for any next step.

A future scenario. If we were to imagine something like an ideal development environment sometime in the next decade, we might include the following approach. The software engineering team is assigned a task to produce a system. Each has an independent software workbench but can send, receive, and share files with all other members of the development team. The statement of work lists the objectives and goals that this program is to satisfy. The team members reduce the objectives to a group of work products and, together with the writer of the objectives, describe the attributes that this work product will have.

The workbench keeps track of the entries and does the "paperwork" of keeping track of inconsistencies, ambiguities, and holes. With the information provided, the designers identify a potentially feasible core for the product to be designed. The prototype of that core is simulated, given to users to manipulate, and exercised by the developers themselves. When a satisfactory prototype is accepted, the contents become the master library of the project manager. The developers use the prototype to search for completed design fragments and string them together. The missing pieces are identified, and a decision is made as to what components should be rewritten and what ones used from the library.

During this time, work effort is monitored by an instrumentation package. It keeps track of the modules and the relationship of those modules to each piece. This tracking ties into the statement of objectives and goals with which the project began. At any design change it will be possible to determine each work product that is attached to that change. Each test that verifies a deliverable is invoked automatically after each change. Each part that has not been used before is examined very carefully, since used parts are always better.

This scenario is not far-fetched. It can and is being enacted today. The difference is the percentage of parts being reused and the integrated tool strategy that ties the components together. The change in approach heralded by the scenario is the rapid decrease in effort associated with the use of reliable pieces. The building-block approach allows the engineer to concentrate on the pragmatism of a project, not the design of the algorithms of the project. Each newly crafted piece must be verified to a level of correctness that is equivalent to the "used" parts.

Software engineers must concentrate on trade-offs.

Engineering is a discipline concentrating on practical trade-offs. Software engineers must also concentrate on trade-offs—not ones associated with the cost of production versus testing, but those of size versus performance and function versus cost.

The role of the science of software development is to come up with the *laws* of software development, the algorithms, transformation rules, and techniques that make new applications possible.

The utility of computers lies in their ability to point out techniques to improve the way we do our work. For the consumer of computational power, the details of programming and software engineering need to be made transparent. For this class of users the process will be eliminated. It will be replaced by software systems designed and constructed by software scientists and software engineers.

To complete the analogy to pyramid building, we should consider structures with the grace of the Eiffel Tower, the size of the World Trade Center, the pleasure of the best of our homes, and the beauty and spontaneity of a sand castle. The principles are the same, the purpose is different, and all derive from a tradition which stretches the creativity of the engineer working in partnership with the scientist.

Software engineering is beginning its rapid trip from its own era of pyramids to the Eiffel Tower. There will be many mistakes along the way, but the direc-

tion is clear. Patience, good will, and the creativity of the software professional will lead to a discipline which will be a *bona fide* member of the engineering community.

Acknowledgments

The author would like to thank the staff of the IBM Software Engineering Institute and the IBM Systems Research Institute for many discussions in the last dozen years. The Institutes have fostered an independent atmosphere and have always encouraged professional involvement by their employees. The author would also like to express his appreciation to the referees who reviewed this paper. Their suggestions were very helpful, and any remaining errors are solely the responsibility of the author.

Cited references

1. R. C. Linger, H. D. Mills, and B. I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley Publishing Co., Reading, MA (1979).
2. C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM* **12,** No. 10, 576–583 (1969).
3. Phillip W. Metzger, *Managing A Programming Project*, 2nd Ed., Prentice-Hall, Inc., Englewood Cliffs, NJ (1981).
4. Randall W. Jensen and Charles C. Tonies, *Software Engineering*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1979).
5. Barry W. Boehm, *Software Engineering Economics*, Prentice-Hall, Inc., Englewood, Cliffs, NJ (1981).
6. M. R. Barbacci, A. N. Habermann, and M. Shaw, "The Software Engineering Institute: Bridging practice and potential," *Software* **2,** No. 6, 4–21 (1985).
7. F. L. Bauer, "Software engineering," *Information Processing 71*, North-Holland Publishing Co., Amsterdam (1972), p. 530.
8. D. T. Ross and K. E. Schoman, Jr., "Structured analysis for requirements definition," *IEEE Transactions on Software Engineering* **SE-3,** No. 1, 6–15 (1977).
9. D. T. Ross, "Structured Analysis (SA): A language for communicating ideas," *Transactions on Software Engineering* **SE-3,** No. 1, 16–33 (1977).
10. M. L. Gillenson and R. Goldberg, *Strategic Planning, Systems Analysis, and Database Design: A Continuous Flow Approach*, Wiley-Interscience, New York (1984).
11. E. Dijkstra, "The structure of 'THE'-multiprogramming system," *Communications of the ACM* **11,** No. 6, 341–346 (May 1968).
12. B. A. Silverberg, "An overview of the SRI hierarchical development method," *Software Engineering Environments*, H. Hunke, Editor, North-Holland Publishing Co., New York (1980), p. 235.
13. D. L. Parnas, "On criteria to be used in decomposing systems into modules," *Communications of the ACM* **15,** No. 12, 1053–1058 (December 1972).
14. C. Bohm and G. Jacopini, "Flow diagrams, Turing machines, and languages with only two formation rules," *Communications of the ACM* **9,** No. 5, 366–371 (1966).
15. H. D. Mills and R. C. Linger, "Data structured programming: Program design without arrays and pointers," *IEEE Transactions on Software Engineering* **SE-12,** No. 2, 192–197 (February 1986).
16. E. W. Dijkstra, "Go to statement considered harmful," *Communications of the ACM* **11,** No. 3, 147–148 (1968).
17. G. Weinberg, *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York (1971).
18. Samuel T. Redwine and William E. Riddle, "Software technology maturation," *Proceedings, 8th International Conference on Software Engineering*, IEEE, Washington, DC (1985), p. 189.
19. J. H. Morrissey and L. S.-Y. Wu, "Software engineering ... An economic perspective," *Proceedings, 4th International Conference on Software Engineering*, IEEE, Washington, DC (1979), pp. 412–422.
20. M. J. Flaherty, "Programming process productivity measurement system for System/370," *IBM Systems Journal* **24,** No. 2, 168–175 (1985).
21. E. Yourdon and L. L. Constantine, *Structured Design*, Prentice-Hall, Englewood Cliffs, NJ (1978).
22. G. J. Myers, *Composite/Structured Design*, Van Nostrand Reinhold, New York (1978).
23. M. A. Jackson, *Principles of Program Design*, Academic Press, Inc., New York (1975).
24. N. Wirth, "Program development by stepwise refinement," *Communications of the ACM* **4,** No. 4, 221–227 (1971).
25. J. D. Warnier, *The Logical Construction of Programs*, Van Nostrand Reinhold, New York (1974).
26. G. D. Bergland, "A guided tour of program design methodologies," *Computer* **14,** No. 10, 13–37 (1981).
27. S. S. Yau and J. J.-P. Tsai, "A survey of software design strategies," *IEEE Transactions on Software Engineering* **SE-12,** No. 6, 713–721 (June 1986).
28. V. R. Basili and R. W. Reiter, Jr., "An investigation of human factors in software development," *Computer* **12,** No. 12, 21–38 (December 1979).
29. D. O'Neill, "The management of software engineering, Part II: Software engineering program," *IBM Systems Journal* **19,** No. 4, 421–431 (1980).
30. M. B. Carpenter and H. K. Hallman, "Quality emphasis at IBM's Software Engineering Institute," *IBM Systems Journal* **24,** No. 2, 121–133 (1986).
31. M. Schaul, "Design using software engineering principles: Overview of an educational program," *Proceedings, 8th International Conference on Software Engineering*, IEEE, Washington, DC (1985), pp. 201–209.
32. J. D. Aron, *The Program Development Process: The Individual Programmer*, Addison-Wesley Publishing Co., Reading, MA (1974).
33. J. D. Aron, *The Program Development Process: The Programming Team*, Addison-Wesley Publishing Co., Reading, MA (1983).
34. M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal* **15,** No. 3, 182–211 (1976).
35. B. P. Lientz and E. B. Swanson, *Software Maintenance Management*, Addison-Wesley Publishing Co., Reading, MA (1980).
36. *Tutorial: Software Cost Estimating and Life-Cycle Control: Getting The Software Numbers*, IEEE, Washington, DC (1980).
37. P. V. Norden, "Curve fitting for a model of applied research and development scheduling," *IBM Journal of Research and Development* **2,** No. 3, 232–248 (July 1958).

38. R. Bazelmans, "Evolution of configuration management," *ACM Software Engineering Notes* **10,** No. 5, 37–46 (1985).
39. *Configuration Control—Engineering Changes, Deviations and Waivers*, U.S. Department of Defense, Washington, DC (1968).
40. *IEEE Standard for Software Configuration Management Plans*, IEEE, New York (1983).
41. W. Bryan, C. Chadbourne, and S. G. Siegal, *Tutorial: Software Configuration Management*, IEEE, Washington, DC (1980).
42. L. A. Belady and M. M. Lehman, "A model of large program development," *IBM Systems Journal* **15,** No. 3, 225–252 (1976).
43. M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE* **68,** No. 9, 1060–1076 (1980).
44. *Computer* (Special Issue: Ada) **14,** No. 6 (1981).
45. G. Booch, *Software Engineering with Ada*, Benjamin/Cummings, Menlo Park, CA (1983).
46. *Computer* (Special Issue: The DoD STARS Program) **16,** No. 11 (1983).
47. R. A. Radice, "Productivity measures in software," *The Economics of Information Processing, Volume 2*, R. Goldberg and H. Lorin, Editors, Wiley-Interscience, New York (1982).
48. Barry W. Boehm, "Software engineering—As it is," *Proceedings, 4th International Conference on Software Engineering*, IEEE, Washington, DC (1979), p. 11.
49. *Information Technology R & D Critical Trends and Issues*, U.S. Congress, Office of Technology Assessment, OTA-CIT-286, Washington, DC (1985).
50. D. Teichroew and E. A. Hershey III, "PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems," *IEEE Transactions on Software Engineering* **SE-3,** No. 1, 41–48 (1977).
51. M. Chandrasekharan, B. Dasarthy, and Z. Kishimoto, "Requirements-based testing of real-time systems: Modeling for testability," *Computer* **18,** No. 4, 71–78 (1985).
52. F. P. Brooks, Jr., *The Mythical Man-Month*, Addison-Wesley Publishing Co., Reading, MA (1975).
53. W. S. Humphrey, "The IBM large-systems software development process: Objectives and direction," *IBM Systems Journal* **24,** No. 2, 76–78 (1986).
54. T. DeMarco, *Controlling Software Projects*, Yourdon, Inc., New York (1982).
55. C. V. Ramamoorthy, W.-T. Tsai, T. Yamaura, and A. Bhide, "Metrics guided methodology," *Proceedings: COMPSAC 85*, IEEE, Washington, DC (1985), p. 111.
56. R. A. Radice, N. K. Roth, A. C. O'Hara, Jr., and W. A. Ciarfella, "A programming process architecture," *IBM Systems Journal* **24,** No. 2, 79–90 (1985).
57. *IEEE Transactions on Software Engineering* (Special Issue on Software Reusability) **SE-10,** No. 5 (1984).
58. Y. Matsumoto, "Some experiences in promoting reusable software: Presentation in higher abstract levels," *IEEE Transactions on Software Engineering* **SE-10,** No. 5, 502–513 (1984).
59. J. M. Neighbors, "The Draco approach to constructing software from reusable components," *IEEE Transactions on Software Engineering* **SE-10,** No. 5, 564–574 (1984).
60. G. Ringland, "Software engineering in a development group," *Software Practice and Experience* **14,** No. 6, 533–559 (1984).
61. James Martin, *Fourth Generation Languages*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1985).
62. B. W. Boehm, "Prototyping vs. specifying: A multi-project experiment," *IEEE Transactions on Software Engineering* **SE-10,** No. 3, 290–303 (1984).
63. M. T. Harandi, "Applying knowledge-based techniques to software development," *Perspectives in Computing* **6,** No. 1, 14–21 (1986).
64. R. C. Waters, "The programmer's apprentice: A session with KBEmacs," *IEEE Transactions on Software Engineering* **SE-11,** No. 11, 1296–1320 (1985).
65. V. R. Basili, R. W. Selby, and D. H. Hutchens, "Experimentation in software engineering," *IEEE Transactions on Software Engineering* **SE-12,** No. 7, 733–743 (1986).

Robert Goldberg *IBM Corporate Technical Institutes, 500 Columbus Avenue, Thornwood, New York 10594.* Dr. Goldberg is an IBM Software Engineering Institute Consultant. Before assuming his current position he was a Senior Institute Instructor at the IBM Systems Reseach Institute, where he taught and developed courses in the areas of software engineering, office automation, information systems architecture, and text processing and retrieval. He joined IBM as a systems engineer in 1968 in a branch office in New Jersey, where he was associated primarily with higher-education accounts. In 1973 he joined the faculty of the Systems Reseach Institute and in 1983 moved to his current position. He is co-editor with Hal Lorin of *Economics of Information Processing, Volume I and II.* He is coauthor with Mark Gillenson of *Strategic Planning, Systems Analysis, and Database Design—A Continuous Flow Approach.* Dr. Goldberg received his B.S. in physics from the Polytechnic University of New York in 1960, and his Doctorate in physics from Rutgers University in 1969.

Reprint Order No. G321-5279.