Professional Software Engineering

W. M. McKeeman

(Editor’s Note: This address was delivered before the 1984 graduating class of the Wang Institute of Graduate Studies in Tyngsboro, Massachusetts. McKeeman, a professor in the School of Information Technology at the institute, is a member of the Software Editorial Board.)

I welcome the 1984 graduates on behalf of the Faculty to celebrate their attainment, certification, and start in practice, of the Mastery of Software Engineering. Whatever individual future each of you finds, you will all be bound by having appeared here together today. I am interested in exploring the significance of that bond.

It is our wish that the bond be professional—that you graduates will be known for your skills and that the regard of the public for you will forge and reforge your mutual ties. It is the combination of mutual regard and public regard which builds a profession.

In my high-tech approach to gathering thoughts for you today, I asked our librarian to computer-search for recent articles on professionalism. I thought it might be helpful to sample the wisdom of more established fields. The search revealed 276 recent articles on the topic. This in itself reinforced my determination to discuss professionalism as an issue, since it is of current importance to so many authors.

The Oxford English Dictionary gives some useful points of departure. I shall paraphrase briefly.

PROFESSION

A vocation in which a professed learning or science is used in its application to the affairs of others. Especially Divinity, Medicine, Law and the Military. Now usually applied to an occupation considered to be socially superior to a trade or handicraft, but formerly, and still in vulgar (and humorous) use, including these.

(1576) Why do you not apply yourself to some kind of profession, or other, wherein there is a certainty, and stay of losing?

(1601) Being Mechanicall, you ought not to walk upon a laboring day without a Signe of your Profession. Speake! What Trade art thou?

We read that learning is essential—that it must be applied to the affairs of others—that a mere trade might have pretense to profession—that a profession achieves personal security—that visible signs of membership are expected.

The fundamental quality of a professional is skill—in a situation where the client is at risk. The skill is, furthermore, based on a learning deeper than any except the elect attain.

Not all who aspire to professionalism achieve it. It is left to the profession to set its standards. You graduates will do this informally when you do, or do not, recommend a colleague for a task. Just as you have learned the team approach to software development, you will find the team approach applies to maintaining your profession. Their successes will be, in part, yours. Your collective success defines your profession.

Clients usually have the option, not whether to consult a professional, but rather which professional to consult. Nor are they always happy with this state of affairs. They may subscribe to the sentiment of Dick the Butcher: “The first thing we do, let’s kill all the lawyers.” Since professions have, to a large extent, a captive audience, they are at pains to ensure a visible trustworthiness.

One way professionals engender trust is by accepting a calling higher than loyalty to any human institution, be it the government or an employer. There are things a professional must do, and things a professional must not do, to remain professional. A Doctor of Medicine subscribes to the Oath of Hippocrates—the well-being of the patient outweighs all other considerations. A lawyer applies his best abilities to the defense of the meanest criminal. A psychiatrist will not divulge secrets learned from his patient. The concept of a professional Software Engineer raises the question of what special obligations we must accept.

Our clients often need systems that are very complex, measured by the standards of other manufactured objects. There are systems of such complexity that fixing one error introduces, on the average, more than one new error into the system. Such a system has reached critical complexity—in the same sense as that of nuclear critical mass. The behavior of such a system can only get worse. It is the responsibility of the Software Engineer to recognize a situation of critical complexity, explain the risks of proceeding to those needing to know, and—where human welfare is at stake—refuse to contribute to making it worse.

Each of you have increased the depths of complexity into which you can safely plunge. And you have found ways to show others—not how to do it, but rather how you shall do it. Like all scientists, your reputation depends on how closely your results correspond to your predictions.

One of the difficult tasks of Software Engineers is to separate the personal joy of exploring new ground from the client’s need for results. Research and development, an inherently risky mode of operation, is often seen as standard operating procedure. But creativity is our most precious asset. It is wasteful to reinvent and reimplement. Your task is to continue to find ways to make building systems ever more routine, and then ways to make the routine effort an efficient use of your time—freeing yourselves to apply the ingenuity from which your title is derived to things that are genuinely new.

The interests of people who pay for systems and the people who use those systems do not always coincide. This tension is properly relieved by political means—it is up to the mechanisms of government to protect society as a whole. A Software Engineer, like any other citizen, owes allegiance to the law above allegiance to employer.

A professional, however, is often exempt from the law—not because it does
not apply, but rather because it is too difficult to apply where the fact of compliance is deeply immersed in the subtleties of practice. At the limits of skill, an outsider cannot discriminate between a blameless mistake and intentional error. The difference is in the mind of the professional. It is for this reason that a professional is called upon to exhibit ethical behavior in every aspect of life—for visible unethical behavior in other things leaves suspect the invisible professional decisions.

Each of the graduates might consider to what extent Software Engineering has reached this state of professionalism. Does the public perceive us as unwilling to attempt tasks we cannot do well? Does the public perceive that our highest loyalty is to the public well-being? In the bluntest of terms, will they drive a car across a bridge built to our software standards? Behind that question is the reason Wang Institute was founded, and the reason you have studied here. We must make the public answer "yes."

Grimes, Miller, Davis Join IEEE Software Board

IEEE Software Editor-in-Chief Bruce Shriver has announced the appointment of Jack Grimes, Edward F. Miller, and Alan M. Davis to the magazine’s Editorial Board.

Jack Grimes comes to the Board with several years of experience, as technical editor of Computer and board member of ACM Computing Surveys and IEEE Computer Graphics and Applications. Most recently, he served as guest editor for a special two-part issue of IEEE CG&A on human factors. Grimes is a systems architect at Intel Corporation in Santa Clara, California. Prior to joining Intel, he was a research director at the ITT Advanced Technology Center in Stratford, Connecticut, responsible for work on office automation for software development in large projects.

Grimes, who received a BS, MS, and PhD from Iowa State University in electrical engineering and computer science, and an MS in experimental psychology from the University of Oregon, has research interests ranging from user interfaces to concurrent systems architecture to management science. For the last five years he has chaired a Siggraph tutorial on the psychology of user interfaces.

A member of the Computer editorial board, Edward F. Miller also has a record of service to Computer Society publications, as a reviewer, referee, and as the guest editor of a special Computer issue on software testing and validation techniques. Miller is the founder, president, and technical director of Software Research Associates, an advanced software technology consulting and development firm. Previously he was the director of the Software Technology Division of Science Applications, Inc.

Miller received a BSEE from Iowa State University, an MS in applied mathematics from the University of Colorado, and a PhD in electrical engineering from the University of Maryland. He is interested in all aspects of software engineering, including program analysis systems, software testing and validation technology, automated software engineering tools, and software maintenance technology. He is currently working on a textbook on software quality assurance.

Alan M. Davis is manager of software engineering for BTG, Inc. Previously, he was director of research and development for GTE Communication Systems, responsible for hardware, operating systems, and system recovery, and director of software technology at GTE Laboratories. He has published numerous papers in software engineering and VLSI. He areas of research interest include VLSI design methodology and tools and software engineering, particularly requirements specification techniques. Davis is chairman of the IEEE Working Group on Software Requirements Standards and a member of the ACM Computing Practices Advisory Panel. He received his BS in mathematics from SUNY, Albany, and his MS and PhD degrees in computer science from the University of Illinois.

Munson named ISSI operations chief

Software productivity expert and IEEE Software Editorial Board member John B. Munson has been named president and chief operating officer of International Software Systems, Inc.

Munson has previously worked on a number of government studies regarding technological approaches to enhancing software productivity. Formerly vice president of technical operations at System Development Corporation, he is currently a member of the US Air Force Scientific Advisory Board and has recently completed a study for the Air Force on the costs and risks associated with mission-critical software.

International Software Systems is engaged in the development of advanced technology and products related to improving software engineering productivity. Its clients include US government and commercial customers as well as government agencies outside of the United States.
The National Science Foundation, through its Computer Science Coordinated Experimental Research program, has funded the Adonis project at SUNY Stony Brook to investigate the potential of a data-oriented computer system within a network environment. This data-oriented network system will exploit the conceptual simplicity of the relational data model to benefit end-users and developers of applications systems by providing a high level, uniform view of data. The system's facilities will include a relational operating system environment, a relational editor, support for logic programming, and a software development environment. Applications of this facility to graphics, VLSI, office automation, and natural-language processing will be explored.

The data-oriented system will be supported by a conceptual data level, in which relations constitute the primitive objects. The system will be implemented on a network of computers, including special hardware to support logic programming and database access. The network software will provide location-transparent access to distributed relations, and operating systems support for distributed programs. The data-oriented network system will be a testbed for research into such issues as resource allocation, concurrency control, and multicast communication.

Data independence is an important principle of computer science research. It is concerned with separating implementation from functionality. In the area of database systems, it embodies the idea that the way in which data is stored can be different from the way in which the user understands and manipulates it. In programming language research, the same concept emerges in the notion of abstract data types: the specification of a data type is kept separate from its implementation. In operating systems, the concept is expressed via synchronization modules such as monitors. The desire for transparent distribution of computation over many processors simultaneously motivates our quest for data independence in a network system.

The thesis advocated here is that relations, and not files, should be the basic data type supported by the system. Relations have a much richer algebra than do files. In many cases the more powerful data manipulation commands associated with relations provide a new way of interacting with the computer. Since this representation provides a conceptual view of data, the corresponding semantics are evident to both users and the system. Thus instead of seeing a memo as a string of characters, the user sees a tuple with address fields and a content field. Meanings are now apparent because structure has been added to the character string. Furthermore, changes to relations or tuple values can immediately trigger system action, since the system can be sensitive to data semantics as well as to syntax. For example, using a relational model of data, one might represent outgoing messages as tuples in the mailbox relation: a user sends mail by adding a tuple to the relation. Similarly, priority may be an attribute of the process relation: a user changes the priority of a process directly by changing the value of that attribute in the tuple associated with that process. Such interpretation of changes to the data is difficult when data is treated simply as a character stream; command intervention is necessary to provide the additional semantics in this case, for example, to change the priority of a process and to change the maximum allowed CPU time. For a system organized as proposed here, however, the emphasis shifts from a command interface to a data manipulation interface. We call an environment of this type data-oriented.

The proposed research seeks to unify the concepts behind data-oriented systems and to expand their domain of applicability. We expect the following advantages to this approach.

1) Since the underlying system manages storage details, applications become simpler to write and maintain. Moreover, data can be shared more easily between applications, since they need not previously agree on physical storage formats.

2) A data-oriented system should be easier for a user to understand, since the structure of the data itself can provide the user with much semantic information. The operations to be applied to data are often straightforward once its structure is known.

3) Since the data has semantic structure it will be straightforward to develop routines that automatically take action in response to changes in the data.

4) Since the system has semantic information about program input and output, it can try to optimize commands. For example, the Unix shell command

```
who|awk \\
```

prints process information about the user on terminal number 7. The command who basically examines the file of all users and computes process information for each one; awk then selects the appropriate one. An obvious optimization strategy is to select the tuple in the user file having terminal number 7, and then compute that process information only. If the command were expressed as a data manipulation query, then standard relational query optimization techniques could be used.

5) The information content of an object can be distinguished from its display format. As an example, consider processing a table of data. An editor can use a form-based view to build the table; a communication system can transmit it by treating the data as a byte stream and ignoring its structure; and a graphics device can output it using a display-based view, which turns the table of data into a pie chart.

The concept of a data-oriented system appears to have enormous scope and flexibility. It provides a uniform interface to all system and user information. The character-stream based Emacs editor on Unix systems does, of course, offer a uniform and extremely powerful interface to character files. However, because of its weak underlying data structure, it is strongly command-oriented. An editor based on relations, which are highly structured objects subject to a powerful set of general operations, can be more flexible and easier to use. The Xerox Star network offers many of the capabilities of a data-oriented system, but its underlying model, which is that of a business office, limits the scope of the operations that can be formed easily. Had it instead been based upon a relational model, it might have been more uniform, simpler, and more easily formalized.

A user or an application in any distributed system must be able to share information with others and have access to global resources. State-of-the-art systems feature localized facilities for each user with access to global resources through a network. It should not affect the user's conceptual model of the system if these resources are local or distributed over the network. Thus the conceptually centralized view of data
that we wish to provide must be built over a distributed database. The forces that lead to data distribution are efficiency (data should be local to the site where it is used most often), authorization (users want tangible evidence that they own the data they create), size (the sheer volume of data may call for its distribution), and reliability (distribution and replication can protect data from individual site failures).

Access to distributed data implies distributed computation—cooperating asynchronous processes which reflect the structure of the underlying distributed system. The development of distributed software is, of course, a significant problem in its own right. Because it is parallel and asynchronous, such software is inherently more complex than the conventional variety. Abstraction techniques, which enable designers of distributed programs to view an underlying network as a single entity, are being developed to deal with this problem. These entail the use of distributed languages, new systems utilities (e.g., loaders, debuggers), and operating system mechanisms supporting location-transparent access to resources. In addition to modeling the asynchronous computational environment, distributed languages ought to reflect the communications facilities of the underlying hardware system, for example, by high-level support for broadcast communications. The development of such tools is a research goal which is directed towards providing a “unified system view.”

Software laurels go to the user-friendly, reports say

In two reports assessing market forces at work in the software field, Input, the Mountain View, California research firm, predicts strong advances for those companies that stress “user-friendliness,” not only in the products they sell but in their corporate style as well.

The first report, entitled Future Skills Requirements for Software Development, analyzes a truth which the organization says has been a long time in coming to information systems managers: if a company has a system analyst who is technically brilliant but who cannot work with people, then his technical skills are essentially useless.

As a result, the report says that IS management has begun to look for systems analysts with qualities that would have been considered irrelevant just a few years ago—interpersonal communications skills, teaching skills, business acumen, sales ability, and the ability to negotiate.

Because training in the humanities is believed to foster these kinds of people, the tables may now be turning on those with strictly technical backgrounds. “The systems analyst of the future may have to be as familiar with the ‘fearful symmetry’ of Blake as with the odd algebraic symmetry of Boole,” the research firm claimed.

The report also forecasts the development of the software environment, and projects the kinds of software (and non-software) skills corporations will have to hire into their IS departments during the next few years.

The core of the report is an analysis of skills requirements and sources for software development in six categories:

- prototyping,
- information centers,
- end-user developed systems,
- applications/report generators,
- applications software packages, and
- traditional software development.

In each category, the report assesses some of the pressures on the IS department and give recommendations on how it should react. The report also investigates staffing and recruitment issues, how to set up career paths within the IS department, and some of the overall effects of decentralized software development.

The second report, entitled Trends and Opportunities in Fourth-Generation Languages, examines the fourth-generation language market with analyses of who makes the decision to buy them, how they are used, what their perceived strengthes and weaknesses are, how big the market is now, and forecasts of how it will grow over the next five years. According to Input, FGLs, with their promise of increased employee productivity and decreased software development backlogs, have generated intense vendor interest.

In addition to defining the FGLs and examining the environments in which they are used, the report also gives an overview of current products, how they are functioning now, and how they could be used in the future.

The report ends with recommendations on how FGLs can be introduced into mainline and production applications, thus expanding their present role as tools to be used for small, decision-support systems.

Both reports are available at a cost of $750 each. For further information, contact Input, 1984 Landings Drive, Mountain View, CA 94043, (415) 960-3990.

Software for kids shown at Boston Museum of Science

Consumers will be able to use computer software programs for free under a program instituted by the Boston Museum of Science. Among the companies donating software to the museum is Random House, whose new line of programs, featuring the famous Peanuts cartoon characters, is oriented toward children pre-school age up to 12 years old.

The Random House software programs will be available for public review at the Hardware/Software Resource Center, which is part of ComputerPlace at the museum. Initially opened as an educational facility for teachers, children and the general public, the Hardware/Software Resource Center offers a place where the public can come in and preview home software products that are available on the market. This will be the largest facility of its kind on the eastern seaboard.

Specific programs in the Peanuts series, all of which run on the Apple II family and Commodore 64, include Charlie Brown’s ABC’s, a pre-school letter-recognition program; Peanuts Maze Marathon, randomly-drawn mazes with animated graphics; Peanuts Picture Puzzlers, an animated puzzle program; Snoopy’s Skywriter Scrambler, a word-recognition program; Snoopy to the Rescue, an arcade program with mathematical challenges; and Snoopy’s Reading an introduction to word families.

The Peanuts gang has invaded the computer software field, as this program on display at the Boston Museum of Science shows.
Japanese conquer software production through labor division

Ware Myers, Contributing Editor

The Japanese seem particularly apt to separate well the two activities—preparation and execution—necessary to do any job of even modest complexity. L. A. Belady told a Compon Fall 84 panel session on Japanese software production methods. This they do both in the sense of subdivision of labor—there are planners and there are implementors—and in the sense of setting aside ample time to design a process, beginning operation only after it looks efficient and satisfactory from all points of view.

Belady, back in the States after a year and a half in Tokyo, observed this method both in everyday life and in various Japanese work environments. Applied to software production, the result is efficiency and high quality. But the sharp separation of planning and execution does reduce flexibility.

"Once in execution mode, there is rarely any chance or even motivation to improvise," he said, and the executors "do not handle exceptions well."

"Nevertheless, the undisciplined behavior of programmers, often observed in even large United States software projects and justified by creativity, is definitely absent," he concluded. In Japan there is a surprising amount of time for innovation and detailed discussions in the first phase, but room only for uncompromising implementation in the second phase.

Following Belady, several speakers from Japanese corporations spoke on their firms' software practices.

Fujitsu. Kaname Kobayashi noted that Fujitsu, which began some software production 30 years ago, restructured its software organization in the early 1970's in response to users' requirements for high quality and reliability. In 1979 it established a division called the "software factory."

Kobayashi cited some interesting figures related to this development. More than 50 percent of the people working in Fujitsu's software operations have been there for more than 10 years, he said. In the most recent two years the cost per line of code has decreased 30 percent. The average number of bugs found within six months after shipment of code has been reduced to one-tenth of the former number in the last five years. The development of software tools accounted for 10 percent of the division's budget. The number of programs converted per man-month has increased by a factor of five in the last five years.

The number of newly developed lines of code per man-month has increased at the rate of about 15 percent per year in the same time. One reason for this increased production, Kobayashi said, has been the provision of a computer terminal for every two persons.

Hitachi. With 14 years of empirical study of the software development process behind it, Hitachi Software Engineering believes that it is now better able to manage cost, schedule, and quality, Satoshi Sakashita reported. Earlier the company had 300 percent cost overruns and years-long delivery delays because of "poor scheduling and lazy monitoring of the project," he noted. Using a cost-progress diagram, Hitachi now monitors progress against plan, detecting problems earlier. As a result, "we have rescued numbers of projects from serious delays and extreme overruns," he asserted.

Running counter to the common US view of Japanese practice, Hitachi has established an independent quality assurance authority that is superior to the producing organizations. "No program can be considered a product until it has been approved by this authority," Sakashita stated. The company has also established ways of monitoring quality during the development process.

Toshiba. Almost half of the lines of code shipped to Toshiba customers each month are reused from earlier software modules, Yoshihiro Matsumoto reported. The organization stores descriptions of modules, interfaces between modules, and traces between interfaces in an archive system, both locally and corporate-wide. A group of designers in each unit manages the codes and documentation of potentially reusable software. This stored information is updated as changes are made and is available to designers both at the factory level and at the company level. The decision to reuse a module is made in a project-level meeting in an early phase of the design cycle.

About one third of the stored modules are smaller than 200 lines of code, about one quarter are between 200 and 1000 lines, another third fall in the 1000-to-10,000-line range, and 10 percent are larger than 10,000 lines. About 45 percent of the modules cover procedures, functions, or subroutines; 25 percent are software tools; and the balance are subsystems and runtime utilities.

Small computers enhance professional productivity

Ware Myers, Contributing Editor

"The small computer is becoming a revolutionary tool in the pursuit of professional productivity," said Harvey G. Cragon at Compon Fall 84 in a speech accepting the Emmanuel R. Piore Award. Cragon, now a faculty member at the University of Texas, won the award for work performed at Texas Instruments where he was a staff member for 25 years.

Cragon divides the development of computers since the early 1940's into three generations. He called the first the "engineering generation," because of the early need for "copious calculations" in engineering. The hallmark of the second generation, beginning in the mid-1960's, was machine productivity. During this period, researchers sought to satisfy the need for "cost-effective and low-cost calculations," he explained.

Now the third generation is beginning to enhance "the productivity of the user rather than the productivity of the machine," he continued. Based on the achievements of the first two generations, third-generation computers can afford to devote many machine cycles to tasks that reduce the load on the user.

Cragon finds the seeds of the third generation in the work of a group of researchers at Xerox Palo Alto Research Center in California. They developed "concepts in interactive computing that are significantly influencing the people-productivity of computers today," he said. These concepts include bit-mapped graphics, high screen-resolution, and the use of a Lisp dialect in personal computing. The Xerox researchers seemed to understand, back in the mid-1970's, that the rapid reduction in the cost of computer cycles was paving the way for machines that would put people first.

Cragon does not believe that the third generation is the result of hardware advances only. He pointed to the striking productivity gains that are possible by using spread-sheet programs such as Visicalc on personal computers. "My experience has been that I would not tackle the work I do in Visicalc in one of the more conventional programming languages. It is not that I cannot do it; the problem is the length of time it takes me to do it." The concept of the program generator, illustrated in Visicalc, is having a profound effect on productivity, he concluded.