CENTER FOR GLOBAL, INTERNATIONAL & REGIONAL STUDIES
University of California, Santa Cruz • College Eight • Santa Cruz, CA 95064 http://www2.ucsc.edu/cgirs
831.459.2833 (Voice) • 831.459.3518 (Fax) global@cats.ucsc.edu

Software Development: A View from the Outside*

Kyle Eischen
keischen@cats.ucsc.edu

*A later version of this paper was published as
"Software: An Outsider's View"
in *Computer* May 2002 (www.computer.org/computer)

CGIRS Working Paper Series
Center for Global International & Regional Studies
University of California, Santa Cruz
http://www2.ucsc.edu/cgirs

WP# 2002-3

Software Development: A View from the Outside*

*A later version of this paper was published as
"Software: An Outsider's View" in *Computer* May 2002 (www.computer.org/computer)

Summary

From the outside looking in, software development debates seem to be thriving. Both software engineering and craft-based method advocates appear no closer to a consensus now than thirty years ago. Considering the debate from a social and economic viewpoint helps reframe the issue, moving towards a clearer understanding of software and software development. There is much that is unique in software, but also much that, especially conceptually, is and has been debated, discussed and learned before. This places software development debates within historical and current perspective. Bringing software into non-technical frameworks involves translating software issues into equivalent social and economic problem domains. The very act of translating and viewing software through social science domains raises questions — What is software? Why is it hard to make? Why do we care? — that normally are assumed by insiders as obvious and well understood. However, such assumptions are often the central issue at stake that prevents solutions from forming.

While there is general agreement on the goal of producing high-quality, low cost software, underlying assumptions concerning the rationalization of software, managing communication and the role of domain-knowledge remain at the heart of development debates. Developing new analogies outside of the current dichotomy of craft and engineering can help highlight these central issues. This also enables software development to learn from existing research on innovation, intellectual work, and information industries to bring new tools to an old debate. Placing domain-knowledge and design at the center of software discussions, opens up a series of issues that can guide software development questions in the short and long-term future.

Outside Looking In

The debate surrounding competing software development methods seems to be thriving [1][2]. Advocates of both software engineering and craft-based (or agile) development approaches appear no closer to a consensus now than thirty years ago when the debate began. Yet from the outside looking in, the debate and the issue of software development itself take on a different perspective. Particularly from a social and economic viewpoint, the importance of the discussion is evident in how widespread the issue actually is. Though not always explicit, how software is made is a central issue in both technical circles and in general social debates on privacy, trade, patents, innovation, immigration and security, to name just a few. Building a bridge between the specific technical and broader issues helps reframe the debate itself, moving hopefully towards a clearer understanding of software and software development.

Bringing software into non-technical frameworks involves translating what appear to be unique software (or even technological) issues into equivalent social and economic problem domains. There is much that is unique in software, but also much that, especially conceptually, is and has been debated, discussed and learned before, which is in part why debates on software development have broader social importance. The very act of translating and viewing software through social science domains raises questions — What is software? Why is it hard to make? Why do we care? — that normally are assumed by insiders as obvious and well understood. However, one of the essential lessons learned from the social sciences is that assumptions are often the central issue at stake that prevents solutions from forming. Drawing on these different traditions is a worthwhile exercise, not because it will solve the software development debate (far from it actually), but rather because it may give insiders new tools and insights to move the discussion forward.

A Little History

A simple take on the software development debate is that it is not new nor is it unique. Arguably, this is true both historically and currently. Looking at the debate over the last thirty years, there are clear cycles of alternating strength between craft and engineering approaches to software development. For a debate to exist so strongly for so long, even as the software industry has grown to maturity, signals to an outside perspective that the debate touches on fundamental, and not just superficial, aspects of software.[1]

Much is made of the 1968 NATO conference that defined software engineering [3], but from a sociological point of view, the follow-up 1970 conference and report [4] is far more revealing. The first meeting, focusing on general concerns and management issues, produced widespread agreement around the clear need to prevent a " software crisis". The second meeting, designed to focus on the technical questions of preventing the "crisis" came to be dominated by a "communication gap" between participants, particularly between theory/practice and computer science/software engineering approaches. From its conception as an independent field of activity, software established a basic pattern that continues to this day, with serious unresolved tensions between issues of management, theory and practice.

Questions of software methods, and the answers generally given, tend to focus on aspects of quality, cost and practice. There is generally agreement that software should be produced at the highest quality for the lowest cost (hence, the agreement at the first conference). However, defining, estimating and measuring quality and cost — as well as the methods to produce such results — were and remain very open to question (hence, the disagreement at the second conference). This seems to be the heart of the matter.

The question is not why there is debate on how to produce software at the highest quality and efficiency, but why there is a gap between agreement on the general problem and disagreement on specific solutions. The most likely answer is that there is not even agreement on the general issue of software quality and efficiency, if the matter is pushed towards real specifics. Much the same way that everyone agrees that a code of law is a good thing, without agreeing on which specific laws are needed. This fundamental difference is only obvious when specific details and methods are discussed, and as a result the questions of software quality, cost and efficiency never really get asked or placed within a broader context.

Rationalizing Software

The broader context involves the issue of rationalization. Going back to Adam Smith's advocacy of the division of labor in the "Wealth of Nations" in 1776, rationalizing production has been a proven method to increase quality, lower cost and raise efficiency. The expansion of industrial capitalism, the rise of modern bureaucracy and scientific-management all added to the momentum to create defined, quantifiable, repeatable production and organizational processes. General competition in markets both proved the power of such rational systems as well as pushed their general adaptation. The ideal example of such a process is the modern automobile industry. But the general logic of rational organization and production exists quite broadly in society, from government to public schools to farming.

It is not surprising that the 1968 NATO meeting, at the beginning of a true computer revolution and the height of US industrial manufacturing dominance, produced general agreement on the increasing importance of software in society and the need for an engineered approach to its production. It is also not surprising that the details of a defined, quantifiable, repeatable process could not agreed upon when the majority of world's software (and even computer systems) was still being crafted, by highly skilled professionals. Professionals who were often engineers by training (so not objecting to the software engineering title), but also direct practitioners who were not involved in nor inclined to build software factories.

The problem then and now between the two viewpoints rests on the question of software rationalization. Assuming that software can be rationalized leads to a whole host of additional assumptions that frame the debate on development methods. Rationalization assumes a quantifiable process, maximized for efficiency by a distinct division of labor, with defined inputs and outputs, managed by an effective rule-bound bureaucratic structure. In other words, a process that is capable of being engineered[2].

History is full of conflicts and debates between craft and engineered approaches. Arguably, the industrial revolution in large part was the success of rational approaches to production over small-scale craft-based methods. In a world of scarce resources, producing more with less is a positive

outcome. Within economics there is always a trade off between benefits to society overall and costs to individual producers from new production processes or organization. This is true for free trade, technological change and manufacturing methods. The argument, with strong evidence behind it, is that in the long-term, both individuals and society benefit from increasing productivity, higher quality, greater variety and lower costs.

Given the rationalization of multiple craft industries over the last two centuries —all that at one time were considered impossible to "manufacture" or mass produce — and the resulting benefits in efficiency and quality, it is reasonable to expect that software can and should become engineered also. Equally expected is that producers of software, like all skilled professionals in the past, will resist this process vehemently, swearing that it is impossible to rationalize software, that software is unique and requires skilled training and "un-quantifiable" knowledge. Max Weber's comment from the turn of the 20th century of a world of "specialists without spirit, sensualists without heart; this nullity imagines that it has attained a level of civilization never before achieved" reflects the attitude that skilled-artisans have always felt toward rationalization. However, Weber also recognized that rationality and bureaucracy are essential, and thus inevitable, features of modern life and its benefits, even if flawed [5][3].

Rationalization and bureaucracy, underpinning modern manufacturing, has consistently placed management objectives and goals as the driver of industry development. The more rationalized, the more explicit, a process the more it can be managed, moving control over the process from producers to managers. The very act of quantifying the process, moving it toward a manufactured method, places skill in new tools and techniques, opening up the possibility of replacing skilled professionals with less-skilled workers. Individual resistance to the process isn't sufficient in and of itself to prevent an overall transformation of the industry towards engineered methods.

So from the outside looking in, the question is why hasn't software development been rationalized? Why, even with a tremendous effort to "engineer" the process both from within the profession and the industry overall, is the development of software locked in the same issues as thirty years ago. Why haven't basic industrial patterns — "software industrialization", "software manufacturing", "software engineering", "software assembly-lines" — become dominant within the industry? Answering the question requires understanding software exceptionalism, or in other words, answering the general question of what exactly is software.

What is Software?

Looking at discussions both within and outside the software profession, it becomes obvious that software is difficult to define. Even where discussions are focused on specific issues like software development methodologies, the definition of software that each party has in mind — though assumed to be the same — is often quite distinct. Issues of software quality, skill and professionalism are all open to interpretation. This clearly limits the rationalization of a process, where even a definition of what needs to be rationalized is difficult to generate. Understanding why definitions are hard to generate is, however, a key step to understanding exactly what software is.

One example is the question of software "patentability" or "copyrightableness". The answer is important, because it frames the debate on software methods. Is software development an invention derived through a scientific method, or is it an act of speech and creativity? In other words, is

software an act of engineering or an act of communication? If software is a rational endeavor, then improving quality involves better and more resources: better management, better tools, more disciplined production and more programmers. If software is a craft, then improving quality involves the exact opposite, focusing on less-hierarchy, better knowledge, more skilled programmers, and greater development flexibility.

The fact that in reality software has characteristics of both patents and copyrights helps to explain why debates around software development are so intractable and conditional. It is difficult, especially from the outside looking in, to separate software into categories that are understood and well defined. Our legal system, our assumptions, our experience force software to be one or another, losing an accurate description in the process. Normally, processes, products and industries are relatively separate areas of study. Such assumptions don't work for software. Debates surrounding open-source development are an example of this. In many ways, the basic conflict is between two distinct visions of how software should be made, distributed and rewarded simultaneously. Even questions of quality — the belief that open, transparent, peer-reviewed products have higher quality — don't deal only with the end product itself, but eventually with how software businesses and processes are structured. Half of the problem in the software development debate is that such assumptions are not clearly out in the open or linked together. The other half of the problem is that being adamantly committed to open source for its quality aspects, still leaves possible being adamantly opposed on sound business grounds. In both cases, what the basic patterns of software are remain hidden.

Brooks [6], as early as 1974, stated that merely adding people to a software project didn't increase its productivity. From an economic viewpoint, this is classic phenomenon of diseconomies of scale. At some point, more people trying to use the same tool (or screw in the same light bulb) cause the whole process to slowdown. From a sociological point of view, this is a fascinating case study into the specific limits of software development, because the essential limit is not the number of tools or a lack of organization, but rather communication. It isn't the number of people, but the increase in lines and quality of communication that complicate software development. Yet, this is exactly the motivation behind bureaucracy and rationally managed processes. To create rules and flows of information so that economies of scale can be achieved. So the solution to software development should be just better planning, particularly in defining a project at its inception to accurately define needed resources. However, as outlined above, such defined, managed processes have not come to dominate.

The simple reason is that communication is equally difficult between developers and users at the inception of a project as it is between developers during the process [7]. This isn't surprising to sociologists. Communication is always difficult, mediated by codes, norms, culture and perceptions that are always contextual. What is new and surprising is that software has the characteristics of other mediums of communication [8]. The process of software development, and building basic requirements, is a process of tacit knowledge communication. This explains much of what is difficult in software development. Translating knowledge from one context to another, like translating any language, involves not just basic rules of grammar and syntax, but also issues of meaning and intent that are contextual and subjective.

Sociologists who take as their domain of interest anything and everything social, clearly understand the difficulty in defining and quantifying, particularly universally and over time, anything that involves human interaction, practice or belief. Yet, this is what software development attempts to do

[9]. Broadly, software is essentially an exercise in translating existing algorithms, in nature or organizations or in practices, into digital form. Much of this domain-knowledge is tacit, undefined, uncodified, developed over time, and often not even explicit to the individuals participating in the process. Even more important, such knowledge and practices are dynamic, constantly evolving and transforming. It should not be surprising that modeling such processes is exceedingly difficult, and that such efforts are often incomplete, impractical or unsatisfactory.

The Key Role of Domain-Knowledge

The importance of domain-knowledge seems consistent for all software development, though it is difficult to see at times. How easily a concept is translated is a function of the nature of domain-knowledge itself. Designing a car is something that most people, even children, can do. The concept is widely known and, in essence, socially agreed upon. Asking someone to design how cars are made is more difficult, though most people have a basic concept of a factory. However, asking someone to design the process of designing a car creates far more difficulty. Is there one right way to design? Are there rules? Is it a burst of inspiration, or 99% hard work and struggle, in a team or by a solitary individual? The farther away from broadly accepted and understood domain-knowledge the more difficult it is to translate. And examples such as these are clearly context specific, changing with culture, location, gender and experience. Looking at software development through the importance of domain-knowledge sheds light on much of what is discussed and practiced within software.

- Organizational, development will be structured around and struggle with the demands of communication, both in the initial project design and during its development.
- If defining software requirements is difficult, defining a universal and fixed process also will be difficult. This suggests that development modeled on a rational process or physical principals, will always have limited applicability to software development.
- Because software products often involve undefined domain-knowledge, the more social the development process, the better. Arguably the extensive use of beta-testing and networks of peer-review are both patterns of focusing tacit knowledge.
- Because the end result of software development will be both a defined product and an aspect of a translation process, it will have characteristics of patents and copyrights.
- Questions of quality will be subjective, often on an individual basis, changing over time and place. The social nature of software insures that even measuring quality will be difficult, involving mixed and relative aspects of cost, reliability and “look and feel”.
- How well domain-knowledge is defined — beyond issues of organization, cost and development time — will play a large role in structuring development. Where the processes being modeled are well understood, where the algorithms of daily life are transparent, then software will be more easily created. Arguably, existing well-engineered processes lend themselves to engineered software development.
- Questions of engineering and craft methods are reflections of the assumptions about and experiences with how well defined, and thus translatable, knowledge-domains are.

Overall, the debate on software development methods would benefit from addressing the basic process of design that is inherently difficult, social and domain-knowledge based. Design in any industry is always a challenging intellectual activity. It is even more so in software, exactly because it is almost a pure design process. Software methods like CMM, and the concept of software

engineering, comes out of a government and military tradition where problems and needs are clearly defined. Issues of management, cost and robustness, not how to define the problem domain, are central. Agile or craft methods, come out of university and programmer communities, where solutions to problems evolved overtime, collectively, based on an open-ended transparent process. The important aspect is not that one method is superior, but rather that both methods address the specific demands of translating domain-knowledge. This shifts the debate to consider what software is expected to do, how to design to meet these needs, and what methods and tools support that process.

The Question of Analogies

Part of the problem in debates on software development stems from the analogies used to describe the benefits of specific methods. Both engineering and craft approach analogies have limitations. Engineering methods stem from a tradition where domains are defined by physical laws and defined parameters, not evolving dynamic systems. Craft approaches highlight the centrality of individual producers, but do not necessarily address the needs of one of the central industries of the coming decades. However, software is not unique in being an intellectual, domain-knowledge focused activity and business. There are other examples outside of engineering and craft examples that software can learn from. The question of analogies is even more significant when trying to explain software issues and debates outside of the software community. Analogies, particularly for outside audiences, can be misleading and only partial. However, a well-chosen analogy can serve as model to highlight key questions, assumptions and possible new directions for software methodologies. A few alternative analogies are:

- **Software as University**.
  The university combines highly independent and creative professionals in an overall structure that can be quite bureaucratic. Many IT firms already formally follow a "campus" model. Innovation, communication and creativity are highly valued. Research is structured around peer-review, with dominant ideas evolving and emerging over time. Teaching and research resist rationalization, remaining highly subjective and individually centered. Issues of productivity and quality are always a mix of qualitative and quantitative factors. Admission to the profession is through a long apprenticeship, guided by senior faculty. The training process tends to not respond in real-time, creating cyclical patterns of under and over supply of academics. Standard quality of personnel at the end of the training process is difficult to guarantee and highly individualistic.

  In terms of specific departments, the analogy becomes more complicated. Engineering and Computer Science have traditional been home to software. However, Economics has a long history of quantitatively modeling social activities (e.g. valuing safety or clean air or health), and testing such models against actual experience. Psychology also has a similar tradition, with already strong links between psychologists and computer fields. Sociology has developed extensive tools to model qualitative behavior in society, organizations and individuals, as well as having a strong understanding of the nature of knowledge, belief and practice. On a general level, the university presents a social model where public support for universities is linked to their contribution to public knowledge and benefit. This fits well with a model of software as a social and public activity, but leaves open to question software as a private industry and the nature of privatized knowledge.

- **Software as Hollywood**
  On an industry level, Hollywood is very similar to software. Historically, it has gone through phases of highly centralized production and more distributed phases of product development. Struggles over control of content and distribution have always been central issues. Film is made using highly skilled teams, normally brought together for individual projects, with members individually judged by overall reputation and experience within the industry. People are trained both formally (through film schools) and informally (through apprenticeships or self-learning). Product quality is hard to evaluate, let alone quantify, prior to release in the market. Success is defined both critically (by peer-review) and publicly (through box office sales), with no guarantee that these will match. Critical success does not guarantee longevity in the industry.

  The production process itself is a non-linear, multi-staged process, where the final product is often undetermined until the final stages. Production is highly susceptible to cost overruns and missed deadlines. Large-budgets and film crews are no guarantee of timely production or actual success in final product. The industry allows for a range of large-scale studio production and independent projects, producing a variety of products from one-off masterpieces to long-running television shows (e.g. soap operas). Film is also directly involved in translating and producing culture on a worldwide scale, producing standardized formats that appeal to the widest possible audience. The limitation as an analogy is in the non-interactivity of the final product that is designed to only entertain passively. However, the software gaming industry, already as large-as Hollywood in revenue terms, presents an interesting example of interactive, yet-very Hollywood like, entertainment.

- **Software as Construction**
  Constructing buildings, while seemingly not an intellectual or domain-knowledge activity, is actually a very helpful model. Building takes a variety of final forms from individual housing to large-scale urban commercial projects. Actual building is done by "do-it-yourselfers", independent local contractors, large construction firms, and global engineering companies. There is a division of labor between architects, developers and builders. The line between these activities is often unclear, as the knowledge of actual construction involves a set of skills covering each aspect. However, there are individual licensing practices for most of these levels, often controlled by government or practitioners themselves. Each project reflects local laws, customs and resources, merging them with the demands and limits of the final consumer. Construction is also highly cyclical, with boom and bust cycles for both firms and labor.

  Overall, building is a very complex organizational problem that draws on experience rather than pre-defined rules or methods. Translating consumer demands into workable end projects that take account of local limits and resources is an intensely intellectual activity. The architecture of a building needs to reflect functional and form demands that are often uncodified. How do people live, work and play are key questions for designers that they may not directly have knowledge with. Quantifying the success of a building, its "look and feel" is difficult. Some buildings are mass-produced, focusing on meeting average needs and demands, and able to be built using average skills and materials. Other projects are one-off developments, requiring innovative designs, materials and highly skilled labor. The more unique a project, the more susceptible it is to cost and time overruns, though all construction projects face this possibility.

Each of these analogies is limited in some way. The exercise, however, is not for an ideal fit. The aim is to highlight the assumptions underlying software methods and help locate possible scenarios that are more reflective of software reality. In each of these examples, issues of design, quantifying quality, efficiency, skill and complexity dominate the development process. In each, the success of the final product and the development process are susceptible to very complex interactions that are difficult to control, anticipate and quantify. Thinking through various examples of intellectual, domain-knowledge based work, opens up debates to new models of software development.

Linking the Inside with the Outside

From the outside, a few things seem clear. There is a real challenge in developing high-quality software, consistently, at a reasonable cost. Software is a unique activity that combines creativity, translation, skill and a disciplined method. The various aspects of software as industry, product and process, both push for and simultaneously resist rational, standardized methods. Software as an industry is subject to the same market forces, accounting practices and government oversight as all other industries. Software products combine both functional and subjective aspects that make standard assessment difficult. Even basic issues of security, privacy and "look and feel" are relative and situational, just as they are in other aspects of life. Rationalizing software processes really involves standardizing intellectual work, which is historically difficult and most likely counter productive. However, this raises tremendous challenges for the industry, which is subject to a market that expects rational management and calculation. Overall, thinking through how to "improve" software involves touching each of these aspects.

As our society and economy become more information and knowledge-based, what is demanded of software and its producers will increase. Software is increasingly a central means of producing, storing, transforming and distributing knowledge [8], raises the importance of discussions on software methods and the demand for ever greater numbers of software solutions and professionals. The original articulation of this trend in 1968 by NATO was correct. However, while software has unique characteristics and role to play in society, it is also a part of a broader trend where multiple industries are facing similar challenges around producing intellectual work on a global scale. Software development can learn from the work on the characteristics of "information economies" and intellectual or information industries.

The flexible, regional-based, information-rich, global economy networked together by flows of people, ideas and finance [10] fits extremely well with the basic patterns structuring software as an industry. It is an environment where competition is based on the management, development and control of innovation and knowledge in both products and people [11]. In order to achieve these aims, firms and organizations are structured for learning, innovation and general goal-setting, in contrast to the bureaucratic, fixed models of the past [12]. Certain industries and organizations, like pharmaceuticals, film or the university, are more adaptable to such an environment, exactly because they are structured around basic issues of intellectual production and management. Design, innovation and intellectual work are basic issues in an information economy. The work detailing these trends is extensive and well-documented, including the quantitative and qualitative factors of R&D, productivity, commercialization, entrepreneurship, learning and network organizations, to name just a few.

All of this provides an extensive resource for software development to draw upon, learn from and apply. There is a history of this in software development, but it has usually drawn upon techniques or methods drawn from manufacturing (TQM) or engineering (Statistical Quality Control) that are based on defined processes linked to quantitative tools. These have a place in software development, but they should follow from an understanding of design as the central activity within software. The issue of design changes the tools and the questions. TQM is an important tool for manufacturing cars, but is far less effective as tool for designing them. Even manufactured products suffer from poor industrial design, resulting in product recalls or failure in the market. Generally questions of design, domain-knowledge and specification don't lend themselves easily to pure statistical analysis. Software shouldn't be expected be different than other industries in this regard, but it also doesn't eliminate the possibility for software — as one of the leading design and intellectual activities of our time — to be an innovator in generating new combinations and insight into tools and resources around intellectual work in an information age.

Looking Towards Future Issues

Short-term concerns in software development are clearly focused on cost, efficiency and quality. However, design and domain-knowledge should be at the center of the discussion. Focusing software development discussions in this way highlights certain issues.

- *Software benefits from peer-review.* The question really is when and how. Early in the process, as in an open source model, or late in the process through beta-testing or public review. Private software will continue to be a factor in society, so a key question is the funding of third-party or public monitoring. Universities have an obvious role to play here, especially in term of protecting the "public commons" in terms of security and privacy. But the process should also be democratized, so that general user comments and perspectives can be included, and that trade-offs between cost, quality and appropriateness can be openly understood and chosen.

- *Much is already right with software development.* There is general agreement on many methods that work, usually rules of thumb (test early, test often; perform daily builds; focus resources early on requirements). What should be articulated is why these work, specifically related to the core activity of translating domain-knowledge and managing intellectual work. This will help compare software to existing case studies of other informational industries, as well develop basic concepts that will help make software understandable to other sectors of the economy and society.

- *Competition in the market is essential, regardless of specific development methods.* Emphasis on the fact that monopoly impedes innovation misses an equally essential point that markets can also evaluate cost and quality trade-offs very efficiently. Well-functioning markets will eventually support well-functioning development processes. However, there is much to be learned from studies of the media industry surrounding control over content and distribution. Keeping open distribution channels, particular as software becomes more service-oriented, is key to insuring innovation in not only products but in processes.

- *Build and borrow tools for evaluating and transferring domain-knowledge.* Simple metrics or models that quantify how well a process is defined will in turn help generate estimates of

time and cost. Such efforts will also help define the resources needed to define the domain originally. Simply questions of how well codified is a process, how well such codes transfer to digital form, and how well can individuals communicate such knowledge all can be evaluated building on tools form the social sciences.

- *Design is difficult across the board, and is inherently a buggy process.* This means that software as a pure design activity, will never be perfect, especially as it models and interacts with human activities in a dynamic environment. It will also never be manufacturing, accept in that aspect of manufacturing that is design. The question is what do we expect from software. Tools to not only define but rank requirements are essential. Software development and software products should reflect these trade-offs. The nature of design, as a human-centered process, also means that mistakes, bugs and unforeseen and unintended consequences will happen, just as they do in other industries and sectors of society. Developing mechanisms to correct these problems, and create incentives (like product liability) to avoid them, need to be part of the discussion. The increasing pervasiveness of software in society will make such issues central, regardless if software increases in quality and reliability.

Longer-term issues involve creating support for software development that reflects both the increasing importance of software in society and the unique demands made in transferring domain-knowledge into effective software.

- *Software education should include general skills of communication, social analysis, design processes, teamwork*, both between software developers and non-technical users. This training involves developing the means to analyze, communicate and work within design environments, simultaneously with developing strong software skills. Such general skills that will serve programmers well in multiple capacities and roles over the course of their professional lives.

- *Cross-disciplinary education to create expertise in new areas of domain-knowledge* in order to combine software skill with specific understanding of problem domains. Biology, finance, film and management are all examples of areas where software skills will be needed in the future. Applied projects, especially in non-technical environments (e.g. IT for non-profits or in the developing world), would reinforce these aspects and would benefit from becoming standardized part of curriculums. These measures would most likely increase participation and interest in software as a dynamic, socially engaged profession.

- *Cross-disciplinary training should extend to non-technical fields as well*, particularly in the fields listed above. An understanding of technology generally, and software practices in particular, will help create the common understanding that will facilitate communication around needs and products. There is a serious lack of understanding and appreciation for software outside of the industry and Computer Science departments. This ultimately limits the effectiveness of future managers, financiers, consultants, educators and economists to both see the potential benefits of software and to effectively and successfully implement new software related projects.

- *Increased support for cross-disciplinary research focused on software processes, products and industries.* Cross-disciplinary teams and support between all of the sciences would help

software understand specific issues of process and product innovation, impacts and direction of R&D funding, aspects of public regulation and support, and help provide an overall map of software presently and into the future. Simultaneously, it would expand the general understanding of software, and help key lessons from software development be incorporated into broader disciplines.

The ultimate goal should be to simply and directly raise the profile of software generally, respecting and making explicit its unique structures and important role in society, and creating the training, research and tools that support both. Such efforts move software not only beyond current methodology debates, but help move software closer and more understandably to the world outside looking in.

References

[1] A. Cockburn and J. Highsmith, "Agile Software Development: The Business of Innovation," *Computer*, pp. 120-122, Sept. 2001.

[2] S. R. Rakitin, "Letters", pp. 4, *Computer*, Dec. 2001.

[3] *Software Engineering: Report on a conference sponsored by the NATO Science Committee*, Edited by P. Naur and B. Randell, Garmisch, Germany, 7th to 11th October, 1968.

[4] *Software Engineering Techniques: Report on a conference sponsored by the NATO Science Committee*, Edited by J.N. Buxton and B. Randell, Rome, Italy, 27th to 31st October, 1970.

[5] M. Weber, *The Protestant Ethic and the Spirit of Capitalism*, New York: Routledge, 1992.

[6] F.P. Brooks, *The Mythical Man-month: Essays on Software Engineering*, Addison-Wesley, Reading, Mass: Addison-Wesley, 1995.

[7] P. McBreen, *Software Craftsmanship: the New Imperative*, New York: Addison-Wesley, 2002.

[8] P. G. Armour, "The Case for a New Business Model", *Communications of the ACM*, Volume 43, Number 8, August 2000.

[9] K. Eischen, *Information Technology: History, Practice and Implications for Development*, Center for Global, International and Regional Studies, University of California, Santa Cruz, Working Paper 2000-4, 2000.

[10] M. Castells, *The Information Age: The Rise of the Network Society*, Cambridge, Mass.: Blackwell Publishers, 1996.

[11] P. F. Drucker, *Management Challenges for the 21st Century*, New York: HarperBusiness, 1999.

[12] C. A. Bartlett and S. Ghoshal, *Beyond the M-Form: Toward a Managerial Theory of the Firm*, Carnegie Bosch Institute for Applied Studies in International Management, Working Paper 94-6, 1994

Endnotes

[1] In contrast, by the time the automotive industry had reached its thirty-year anniversary (roughly 1930), the fundamental patterns of production (Ford's assembly line), product (standardized design) and industry (a few dominant national players like Ford and GM) had been clearly established.

[2] The term 'Engineering' originated in 1720 and is "**1**: the activities or function of an engineer **2 a**: the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to people in structures, machines, products, systems, and processes b: the design and manufacture of complex products <software engineering> 3: calculated manipulation or direction (as of behavior) <social engineering>". Source: Merriam-Webster New Collegiate Dictionary.

[3] This is Weber's famous "iron cage". Rationality and bureaucracy are essential to modern society and its material achievements, but limit creativity, innovation and spirituality. This is by definition. Rationality leads to rule-bound, fixed parameters and hierarchies that place control, knowledge and power in institutions and not individuals. Bureaucracy is essentially an institutionalized algorithm that takes general inputs and produces fixed and anticipated outcomes.