Software Development: An Outsider's View

Kyle Eischen
University of California, Santa Cruz

Comparing software with other disciplines and industries can help resolve the perennial debate between those who advocate an engineering approach to development and those who advocate a craft-based approach.

A thriving debate surrounds competing software development methods.[1,2] Advocates of both traditional software engineering and craft-based—also known as agile—development approaches appear no closer to a consensus now than they were when the controversy began 30 years ago.

Yet, from the outside looking in, the debate—and software development itself—take on a different perspective. How developers make software is often a central if not necessarily explicit concern of technical circles and general social debates on privacy, trade, patents, innovation, and security. Building a bridge between the specific technical issues and the broader social concerns can thus help us move toward a clearer understanding of software and software development.

Bringing software into nontechnical frameworks involves translating specific software issues into equivalent social and economic domains. Much in software is unique, but much also has been debated, discussed, and learned before. The act of translating and viewing software technologies and practices through social science domains raises interesting questions: What is software? Why is it hard to make? Why do we care?

Insiders normally assume that the answers to these questions are obvious and well understood. However, social science teaches that unstated or undefined assumptions often are the essential issues in any debate. Drawing on these different traditions offers a worthwhile exercise, not because doing so will solve the software development debate, but rather because insiders may acquire new tools and insights for moving the discussion forward.

A LITTLE HISTORY

The ongoing debate has shown clear cycles of alternating strength between craft-based 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, shows that it touches on fundamental aspects of software. In contrast, other modern industries have resolved these fundamental issues rather quickly. For example, by the time the automotive industry had reached its 30th anniversary, around 1930, the fundamental issues of production, as exemplified by Ford's assembly line; product, as exemplified by standardized design; and industry, as exemplified by the few dominant national players like Ford and GM, had all been clearly established.

Much has been made of the 1968 North Atlantic Treaty Organization conference that defined software engineering.[3] Yet, from a sociological perspective, the 1969 NATO conference and the report[4] it generated prove far more revealing.

The first meeting, which focused on general concerns and management issues, produced widespread agreement about the clear need to prevent a software crisis caused by increasing software complexity and a lack of skilled software professionals.

The second meeting, designed to focus on the technical questions of preventing the crisis, saw a "communication gap" split participants between the theory versus practice and computer science versus software engineering approaches. From its conception as an independent field of activity, software thus established a basic pattern that continues to this day, with serious unresolved tensions between management, theory, and practice issues.

Discussions then and now tend to focus on the quality, cost, and practice aspects of software development methods. As demonstrated at the 1968 NATO conference, the software community generally agrees that they should produce the highest quality software for the lowest cost. However, defining, estimating, and measuring quality and cost—as well as the methods to produce such results—have remained open to question since the second NATO conference.

The central issue is why software developers agree on the general problem but disagree on the specific approaches to solve it. The main answer is that when they get down to specifics, developers disagree on the general definition of software quality and efficiency—much as people will agree that a code of law is good, but they disagree about the specific laws it should contain. This fundamental difference becomes obvious only when developers discuss methods in detail; as a result, they never ask basic or first-principle questions about software quality, cost, and efficiency or place such issues within a broader context.

RATIONALIZING SOFTWARE?

The broader context is software rationalization. Since 1776, when Adam Smith advocated the division of labor in *Wealth of Nations*, rationalizing production has offered a proven method for increasing quality, lowering cost, and improving efficiency.

The expansion of industrial capitalism, the rise of modern bureaucracy, and scientific management all accelerated the momentum for creating defined, quantifiable, repeatable production and organizational processes. General competition in markets proved the power of such rational systems and pushed their general adaptation. Although the modern automobile industry provides perhaps the most well-known example, the general logic of rational organization and production exists quite broadly in society, from government to public schools to farming.

Predictably, then, the 1968 NATO meeting—which took place 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.

Just as predictably, the participants in the 1969 conference could not agree on the details of a defined, quantifiable, repeatable process while highly skilled professionals still crafted most of the world's software. These professionals—often engineers by training and thus not averse to the *title* of software engineer—worked as direct practitioners who were neither involved in nor inclined to build software factories.

Software developers agree on the general problem but disagree on the specific approaches to solve it.

Additional assumptions

Assuming that software can be rationalized leads to a host of additional assumptions that frame the debate around 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—all of which results in a process capable of being engineered.

Arguably, the industrial revolution signaled the rise and dominance 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, a perennial tradeoff exists between the benefits to society overall and the costs individual producers must bear from the introduction of new production processes or organizations. This tradeoff holds true for free trade, technological change, and manufacturing methods. Economists make the argument, with strong evidence behind it, that in the long term, both individuals and society benefit from increasing productivity, higher quality, greater variety, and lower costs.

Engineering software

Given the rationalization of multiple craft industries over the past two centuries—all of which were once considered impossible to manufacture or mass-produce—and the resulting benefits in efficiency and quality, we can reasonably expect that software can and should become engineered as well. We also can expect that software producers, like

Software development debates become intractable and conditional because software has characteristics of both patents and copyrights.

all skilled professionals in the past, will resist this process vehemently, deeming it impossible to rationalize software given its unique nature, which requires skilled training and unquantifiable knowledge to master.

Rationalization and bureaucracy

At the turn of the 20th century, Max Weber foresaw a world of "specialists without spirit, sensualists without heart; this nullity imagines that it has attained a level of civilization never before achieved." His comment reflects the attitude that skilled artisans have always felt toward rationalization. However, Weber also recognized that rationalization and bureaucracy play essential roles, making them inevitable, if flawed, features of modern life[5] that limit creativity, innovation, and spirituality—by definition.

Rationalization 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.

Thus, rationalization and bureaucracy—the underpinnings of modern manufacturing—have consistently emphasized management objectives and goals as the drivers of industrial development. The more rationalized and explicit a process, the more it can be managed, moving control of the process from producers to managers.

The very act of quantifying the process and 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 to prevent an overall transformation of the industry toward engineered methods.

So, we must question why software development hasn't been rationalized. Why, even with a tremendous effort to engineer the process from within the profession and from the industry overall, does software development continue to confront the same issues it did 30 years ago? Why haven't basic industrial patterns—software industrialization, software manufacturing, software engineering, and software assembly lines—become dominant within the industry? Answering these questions requires understanding software's precise nature.

SOFTWARE DEFINED

Rationalizing software development requires first defining software quality, skill, and professionalism. That this is still an open issue after 30 years explains why software development remains unrationalized and unique from other industries or processes. However, understanding *why* arriving at a definition is difficult does provide a key step toward understanding exactly what software is.

Patent or copyright?

Consider for example whether software should be patented or copyrighted. The answer to this question 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?

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

That software has characteristics of both patents and copyrights helps explain why software development debates become so intractable and conditional. It is difficult, especially from the outside looking in, to separate software into understandable and well-defined categories. Our legal system, assumptions, and experience force software to be one or the other, losing an accurate description in the process.

Opening open source

Processes, products, and industries normally involve relatively separate areas of study. Such assumptions don't work for software. The debate surrounding open source development provides an example. In many ways, the basic conflict between open source and proprietary methods stems from two distinct visions of how software should be made, distributed, and rewarded. Even questions of quality—the belief that open, transparent, peer-reviewed products have higher quality—deal not with the end product but also eventually with the structure of software businesses and processes.

When debating software development methods, a central problem arises from not bringing assumptions of quality, organization, and reward clearly into the open or linking them together. For example, being adamantly committed to open source for its quality or organizational aspects still doesn't address questions of business or economic rewards.

Communicating design

As early as 1974, Fred Brooks[6] stated that merely adding people to a software project didn't increase its productivity—a classic diseconomies-of-scale phenomenon. At some point, when more people try to use the same tool, the entire process slows down.

From a sociological viewpoint, this situation presents a fascinating case study into the specific limits of software development. The essential limit is not the number of tools or lack of organization but, rather communication. The increase in lines and quality of communication, not the number of people, complicate software development. Yet, bureaucracy and rationally managed processes are built to create rules for communicating and managing information flows to achieve economies of scale.

Thus, the solution to software development should be better planning, particularly in defining a project at its inception to accurately identify needed resources. As I've outlined, however, such defined, managed processes have yet to become prevalent—exactly because communication is just as difficult between developers and users at a project's inception as it is between developers *during* the process.[7]

Communication and sociology

As sociologists know, communication is inherently difficult, mediated by always-contextual codes, norms, culture, and perceptions. What is new and surprising is that software has the characteristics of other communication mediums.[8] Building basic requirements, for example, involves a process of tacit knowledge communication, which explains much of what is difficult in software development. Translating knowledge from one context to another, like translating any language, involves not just basic grammar and syntax rules, but also issues of meaning and intent that are contextual and subjective.

Sociologists clearly understand the difficulty in universally defining and quantifying over time anything that involves human interaction, practice, or belief—exactly what software development attempts to do.[9]

Broadly, software offers an exercise in translating existing algorithms—in nature, organizations, or practices—into digital form. Much of this domain knowledge is tacit, undefined, uncodified, and developed over time, often without being explicit even to the individuals participating in the process. More importantly, such knowledge and practices are dynamic, constantly evolving, and transforming. It should not surprise us that modeling these processes proves exceedingly difficult and that such efforts are often incomplete, impractical, or unsatisfactory.

Communication is inherently difficult, mediated by always-contextual codes, norms, culture, and perceptions.

DOMAIN KNOWLEDGE'S KEY ROLE

The ease with which a concept translates is a function of how well accepted and defined its specific domain knowledge is. Most children can, for example, design a car using widely known and socially agreed-upon concepts. Although most people have a basic idea of how a factory operates, asking someone to design a process for making cars is more difficult. Asking someone to design the process of designing a car poses an even greater challenge. Is there one right way to design? Are there rules? Is it a burst of inspiration or is it 99 percent hard work performed by a solitary individual or by a team?

Translation implications

The further away from broadly accepted and understood domain knowledge we move, the more difficult translating that knowledge becomes. The issues are clearly context specific, changing with culture, location, gender, and experience. Assessing the importance of domain knowledge to software development can bring new understanding to topics such as the following that developers discuss and practice within the discipline:

- 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, fixed process will be difficult as well. This observation suggests that development modeled on a rational process or physical principles 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, tools like peer-review networks and extensive beta testing focus tacit knowledge.
- Because software development's end result 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 often be individually subjective, changing over time and place. The social nature of software ensures that even measuring quality will be difficult, involving

Overall, this debate would benefit from addressing a basic software design process that is inherently difficult, social, and based on domain knowledge.

mixed and relative costs, reliability, and look-and-feel aspects.

- How well we define domain knowledge—beyond issues of organization, cost, and development time—plays a large role in structuring development. When developers thoroughly understand the processes they are modeling, and when algorithms are most transparent, they can create software with greater ease. Arguably, then, existing well-engineered processes lend themselves to engineered software development.
- The software development methods they choose, and the sides they take in software debates, generally reflect developers' assumptions and experiences regarding how well defined, and thus translatable, knowledge domains are.

Overall, the software debate would benefit from addressing a basic design process that is inherently difficult, social, and based on domain knowledge. Design presents a challenging intellectual activity in any industry, but this maxim is especially true for software, exactly because it involves an almost pure design process.

Design assumptions

Software engineering and software methods like the Software Engineering Institute's Capability Maturity Model arise from a government and military tradition of dealing with clearly defined problems and needs. Issues of management, cost, and robustness—not how to define the problem domain—are central.

Agile or craft-based methods, on the other hand, originate in university and programmer communities, where solutions to problems evolve collectively over time, based on a transparent, open-ended process. Both engineering and craft-based methods address the specific demands of translating domain knowledge, but they differ in their design assumptions and problem definitions. This shifts the software development debate to consider what software is expected to do, how to structure a design process to define and meet these needs, and what methods and tools support that process.

BRINGING SOFTWARE INTO SHARP FOCUS

When debating software development, the analogies used to describe a specific method's benefits can cause confusion. Both engineering and craft-based analogies have limitations. Engineering methods stem from a tradition that describes domains according to physical laws and defined parameters, not dynamic evolving systems. Craft-based approaches highlight the centrality of individual producers, but they do not necessarily address the demands of a global industry.

While the scenarios in the "Software Development Analogies" sidebar are in some ways limited, they don't need to provide an ideal fit. Rather, they highlight the assumptions underlying software methods and help to locate possible scenarios that better reflect software reality.

In each analogy, issues of design, quantifying quality, efficiency, skill, and complexity dominate the development process. In each, the final product's success and the development process itself are susceptible to complex interactions that are difficult to control, anticipate, and quantify. Thinking through various examples of intellectual, domain-knowledge-based work opens the ongoing software debate to the consideration of new development models.

LINKING THE INSIDE AND OUTSIDE

From the outside, a few things seem clear. Developing high-quality software consistently, at a reasonable cost, presents a real challenge. Software combines creativity, translation, skill, and a disciplined method. The various aspects of software as industry, product, and process push for and simultaneously resist rational, standardized methods.

Industrial constraints

Software as an industry must confront 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.

Rationalizing software processes involves standardizing intellectual work, which is historically difficult and most likely counterproductive. However, these obstacles raise tremendous challenges for the industry, whose market expects rational management and calculation. Improving software involves considering each of these aspects.

As our society and economy become more information- and knowledge-based, demands on software and its producers will increase. As an increasingly central means of producing, storing, transforming, and distributing knowledge,[8] software demands ever increasing numbers of solutions and professionals. The original articulation of this trend at the 1968 NATO conference has proven correct.

Design, innovation, and intellectual work are basic issues in an information economy.

Global implications

Software also forms part of a broader trend in which multiple industries face similar challenges to produce intellectual work on a global scale. Software development can learn from work on the characteristics of information economies and intellectual or information industries.

Networked together by flows of people, ideas, and finance,[10] the global economy fits well with the basic patterns that structure software as an industry. This environment bases competition on the management, development, and control of innovation and knowledge in both products and people.[11] To achieve these aims, firms and organizations must be structured to promote learning, innovation, and general goal-setting, thus breaking with past bureaucratic, fixed models.[12]

Certain industries and organizations, like pharmaceuticals, film, or universities, can adapt more easily 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 quantitative and qualitative factors of R&D, productivity, commercialization, entrepreneurship, learning, and network organizations.

Manufacturing legacy

All of these examples provide resources that software development can draw upon and apply. Historically, software developers have indeed done so, but usually in the context of methods like manufacturing's Total Quality Management or engineering's Statistical Quality Control—methodologies based on defined processes linked to quantitative tools. TQM and SQC have a place in software development, but applying these methodologies should follow from an understanding of design as the central software activity.

The design issue changes the tools and the questions. TQM, an important tool for *manufacturing* cars, is far less effective 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. We shouldn't expect software to differ from other industries in this regard. Neither should we ignore the possibility that software—as a leading design and intellectual activity—could be an innovator that generates new combinations and insight into tools and resources arising from intellectual work in an information age.

FUTURE ISSUES

Short-term concerns in software development clearly focus on cost, efficiency, and quality. However, putting design and domain knowledge at the discussion's center will highlight the following issues:

- *Software benefits from peer review*. Is peer review benefical 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, so a key question is funding third-party or public monitoring. Universities have an obvious role to play here, especially in terms of protecting security and privacy. But the process should also be democratized, so that general user comments and perspectives can be included, and tradeoffs between cost, quality, and appropriateness are openly understood and chosen.
- *Much is already right with software development*. Developers generally agree on many methods that work, usually rules of thumb such as test early, test often; perform daily builds; and focus resources early on requirements. We also need to articulate why these methods work, specifically as they relate to the core activity of translating domain knowledge and managing intellectual work. This will facilitate comparing software to existing case studies of other informational industries. It will also help in developing basic concepts that can make software understandable to other economic and social sectors.
- *Competition in the market is essential, regardless of specific development methods*. Emphasizing that monopoly as an impediment to innovation misses an equally essential point: Markets can also evaluate cost and quality tradeoffs efficiently. Well-functioning markets will eventually support well-functioning development processes. However, we can learn much from studies of the media industry's control over content and distribution. Keeping distribution channels open, particularly as software becomes more service oriented, will be key to ensuring innovation in products as well as processes.
- *Build and borrow tools for evaluating and transferring domain knowledge*. Simple met-

rics 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 determine the resources needed to define the domain originally. We can build on tools from the social sciences to evaluate simple questions of how well codified a process is, how well such codes transfer to digital form, and how well individuals can communicate such knowledge.

- *Design, an inherently buggy process, is difficult across the board.* 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, except to the extent that manufacturing is a design activity. We must decide what we expect from software. Tools that define and rank requirements are essential. The nature of design, as a human-centered process, also means that mistakes, bugs, and unforeseen and unintended consequences will occur, just as they do in other industries. Developing mechanisms to correct these problems, and creating incentives to avoid them, must be part of the ongoing discussion. The increasing pervasiveness of software in society will make such issues central, whether software quality and reliability increase or not.

Longer-term issues involve creating development support 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.* These skills include communication, social analysis, design processes, and teamwork between both software developers and nontechnical users. In addition to developing strong software skills, this training should instill the means to analyze, communicate, and work within a design environment.
- *Cross-disciplinary education will create expertise in new domain knowledge areas.* This training will let developers combine their software skills with specific understanding of problem domains. Biology, finance, film, and management all provide examples of areas in which software skills will be needed in the future. Applied projects, especially in nontechnical environments—such as creating information systems for nonprofit organizations or in developing countries—would build expertise and benefit from becoming a standardized part of curricula. These measures would most likely increase participation and interest in software development as a dynamic, socially engaged profession.
- *Cross-disciplinary training should extend to nontechnical fields.* An understanding of technology generally, and software practices in particular, will help facilitate communication about needs and products. Until understanding and appreciation of software extends beyond the computing industry and computer science departments, the ability of future administrators, managers, financiers, consultants, educators, and policymakers to see software's potential benefits and effectively implement new software-related projects will remain limited.
- *Cross-disciplinary research merits increased support.* This work should focus on software processes, products, and industries. Cross-disciplinary teams and closer ties between all scientific disciplines would help developers understand specific issues in process and product innovation, the effects and optimal direction of R&D funding, and aspects of public regulation and support. This knowledge would, in turn, help provide an overall map of software that extends from the present forward. Increased interdisciplinary awareness would also expand the general understanding of software and help incorporate key lessons from software development into broader disciplines.

As software becomes more service oriented, keeping distribution channels open will be key to assuring innovations in products and services.

Our 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 training opportunities, research projects, and tools to support it. Such efforts move software not only beyond current methodology debates but also 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*, Sept. 2001, pp. 120-122.
2. S.R. Rakitin, "Letters," *Computer*, Dec. 2001, p. 4.

3. P. Naur and B. Randell, eds., "Software Engineering: Report on a Conference Sponsored by the NATO Science Committee,"http://www.cs.ncl.ac.uk/peoplebrian.randell/home.formal/NATO/nato1968.PDF.
4. J.N. Buxton and B. Randell, eds., "Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee," http://www.cs.ncl.ac.uk/people/brian.randell/home.formal/NATO/nato1969.PDF.
5. M. Weber, *The Protestant Ethic and the Spirit of Capitalism*, 1920, reprint, Routledge, London, 1992.
6. F.P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Reading, Mass., 1995.
7. P. McBreen, *Software Craftsmanship: The New Imperative*, Addison-Wesley, New York, 2002.
8. P.G. Armour, "The Case for a New Business Model," *Comm. ACM*, Aug. 2000, p. 19.
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, http://www2.ucsc.edu/globalinterns/wp/wp2000-4.pdf.
10. M. Castells, *The Information Age: The Rise of the Network Society,* Blackwell Publishers, Cambridge, Mass., 1996.
11. P.F. Drucker, *Management Challenges for the 21st Century,* HarperBusiness, New York, 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, Pittsburgh, 1994, http://www.gsia.cmu.edu/bosch/bart.html.

Kyle Eischen *is associate director of Regional and Informational Research at the Center for Global, International and Regional Studies, University of California, Santa Cruz. His research interests include innovation and design processes and the social and economic impact of information technology. Currently he is codeveloping the Technology and Human Development Initiative, a UC systemwide "skunk works" for applied technology projects in the developing world. Eischen received an MA in sociology from UC Santa Cruz and an MPIA in international affairs (applied economics) from the Graduate School of International Relations and Pacific Studies at UC San Diego. Contact him at kbe@ieee.org.*