

# UC Santa Cruz

## Working Papers

### Title

The Social Impact of Informational Production: Software Development as an Informational Practice

### Permalink

<https://escholarship.org/uc/item/9rp0c3w4>

### Author

Eischen, Kyle

### Publication Date

2002

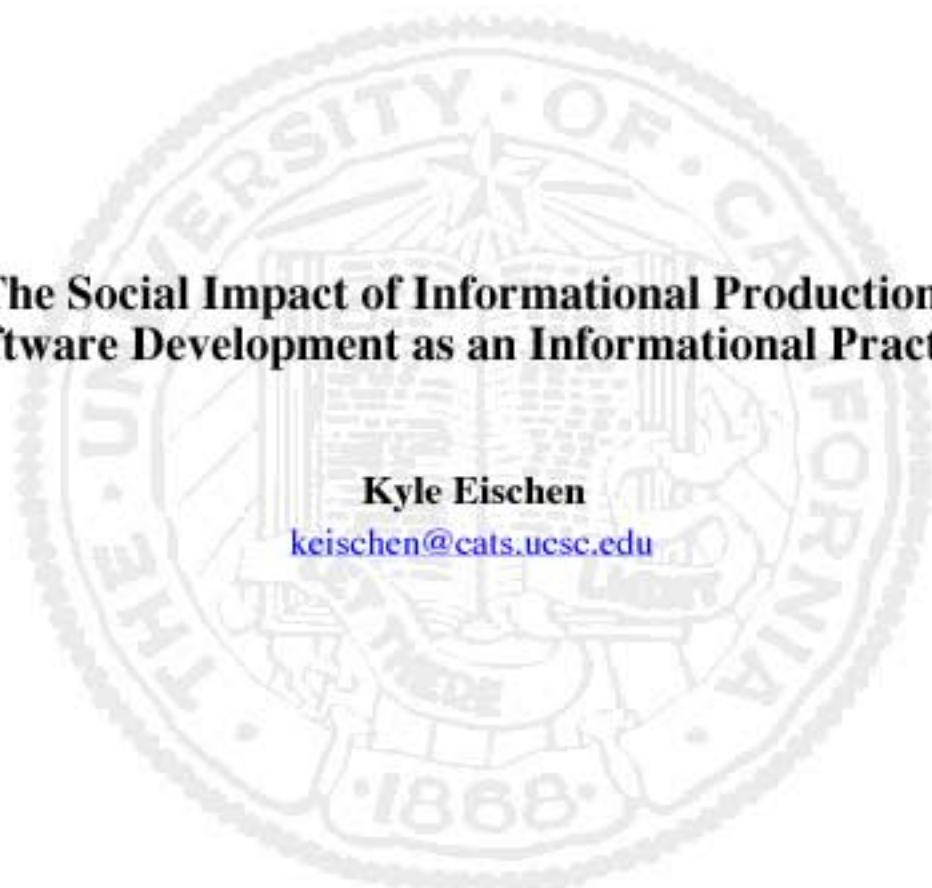
# CENTER FOR GLOBAL, INTERNATIONAL & REGIONAL STUDIES

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](mailto:global@cats.ucsc.edu)



## The Social Impact of Informational Production: Software Development as an Informational Practice

**Kyle Eischen**

[keischen@cats.ucsc.edu](mailto:keischen@cats.ucsc.edu)

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

**WP# 2002-1**

## **Abstract**

Software is a unique process that draws on socially structured domain-knowledge as its central resource. This clarifies the importance of information and design in an informational environment, as well as signals the impact of digital architectures in structuring new patterns of social interaction. These informational patterns are embedded in software both technically and through the development process, resulting in a strong cohesion between production, product and industry structures.

Software increasingly is a central, but hidden factor within the global environment structuring production practices, embedded in products and shaping social interactions. Many of the central conflicts, assumptions and patterns in information societies are directly linked to software's informational practice or highlighted first through software focused debates. Ignoring such patterns directly limits our understanding and analysis of the broader social transformations in the global environment. Failing to recognize these processes limits the space for social debate, policy and action around the establishment and evolution of these new digital architectures at the locus of their development.

**Key Words:** *Software, Rationalization, Domain-knowledge, Information Society, Informational Production*

## **Acknowledgements**

I would like to thank Lisa Nishioka, Del Cielo and the Center for Global, International and Regional Studies for insightful conversations and institutional support without which this work would not have been completed. Additionally, Jon Guice, Paul Lubeck, Manuel Pastor, Helen Shapiro, and Nirvikar Singh all provided essential comments at key stages as this work developed.

## I. The Black Box of Software Development

One of the most significant information technology “black boxes” is software, and its opaqueness has increasingly serious implications for understanding new social tensions and patterns. Software is a central industry within the global economy, and is increasingly surpassing traditional manufacturing as a source for economic growth in both advanced and developing regions. This increasing importance alone — like previous detailed studies of industrial processes in automobiles, electronics or textiles — makes understanding software essential to evaluating key social and economic issues in the coming decades. Perhaps more importantly, the increasing pervasiveness of software within society raises important institutional and social questions that cannot be fully explained or understood without opening software to detailed analysis. Central ‘information society’ issues of privacy, identity, property rights, access, monopoly, speech, trade, organization and innovation are either directly software-driven or highlighted first by software centered debates in society.

While many of these aspects of software are well discussed as parts of the “new economy,”<sup>1</sup> software itself is not well defined or understood either as a technology or as a production process. Within social science analysis, software tends to be approached at a general level, either considering software as one among many information technologies (most commonly as an additive to hardware development), or where specifically considered focused on large-scale industry or product patterns.<sup>2</sup> What exactly software is and how it is produced remains undefined, even unimportant. However, these micro-level aspects of software are central issues within the software profession itself. The specific nature of software and its production is and has been a subject of rigorous debate within the professional literature for decades, but rarely draws upon outside social or economic analysis to help frame these debates or place them in historical or current contexts. An essential aspect of opening software is to bring these detailed micro-level technical and process discussions into broader social science frameworks that strengthen the understanding of both new production patterns and their larger social impact.

Opening software means moving beyond pure technological foci to map the patterns that define software as an informational technology, practice and industry. This makes transparent the

---

<sup>1</sup> F. Cairncross, *The Death of Distance*; P.E. Moody and R. E. Morley, *The Technology Machine: How Manufacturing will Work in the Year 2020* (New York, NY: Free Press, 1999); J.B. Quinn, J. Baruch and K. Anne Zien, *Innovation Explosion: Using Intellect and Software to Revolutionize Growth Strategies* (New York: The Free Press, 1997); L. Lessig, *Code: and Other Laws of Cyberspace* (New York: Basic Books, 1999).

<sup>2</sup> S. O’Riain, *The Flexible Developmental State: Globalization, Information Technology and the ‘Celtic Tiger’* presented at the Global Networks, Innovation and Regional Development: The Informational Region as Development Strategy conference (University of California, Santa Cruz, November 11-13<sup>th</sup>, 1999) <http://www2.ucsc.edu/cgirs/>; B. Parthasarathy *Institutional Embeddedness and Regional Industrialization: The State and the Indian Computer Software Industry* presented at the Global Networks, Innovation and Regional Development: The Informational Region as Development Strategy conference (University of California, Santa Cruz, November 11-13<sup>th</sup>, 1999) <http://www2.ucsc.edu/cgirs/>; R. N. Langlois and D. C. Mowery “The Federal Government Role in the Development of the U.S. Software Industry”, in D.C. Mowery, Editor, *The International Computer Software Industry: A Comparative Study of Industry Evolution and Structure*, (New York: Oxford University Press, New York, 1996); D.J. Hoch, C. R. Roeding, G. Purkert and S. K. Lindner, *Secrets of Software Success* (Harvard Business School Press, Boston, 2000); R. Heeks *Software Strategies in Developing Countries* (University of Manchester: Institute for Development Policy and Management, Working Paper 6, June, 1999) <http://www.man.ac.uk/idpm>.

domain-knowledge, non-material, design process that centers on the production, management and distribution of information. In doing so, it becomes increasingly apparent that the nature of software as a technology, product or as an industry is intimately linked to the structure of its production. Revealing the specific aspects of this production — which information is important, how information is produced and manipulated — helps isolate the social nature and impact of information and knowledge in an information-driven society. In combination, the structure of production and the use of information goes very far in detailing the institutional and organizational trends, tensions, flows and linkages that structure software’s impact within society.

## II. The Technical Structures of Information Technologies: The Historic Development of Algorithmic, Digitalized Information

The simplest definition of information technology is one word: algorithm.<sup>3</sup> Simply defined, an algorithm is a procedural description of how something should be accomplished. Such patterns are widely occurring both in society and nature. Bureaucracies are socially institutionalized algorithms, as are the patterns that govern basic biological processes<sup>4</sup>. Presently, the most widely applied example of this is within information technologies, both in hardware and software processes.

Algorithmic logic was revived as a defined and respectable field of knowledge by Leibniz, the co-creator of the calculus, at the end of the 17<sup>th</sup> century. Leibniz offered a very simply but powerful idea: the entire universe can be described as comprised of god and nothingness.<sup>5</sup> In other words, all of life follows a discrete binary system that can be modeled, or coded, within a logical algebraic framework. As such, real-world processes could be mapped using mathematical symbols, if the underlying algorithms could be identified. This opened the theoretical possibility of modeling both the social processes of bureaucracies and the basic sequence of DNA, among others, as mathematical abstractions.

Leibniz insight into algorithmic logic and binary systems are the conceptual and technical heart of information technology.<sup>6</sup> All information technologies essentially define a logical algebraic function that produces consistent outcomes for specific processes, which is then codified either as software or hardware formats. “The distinguishing feature of an algorithm is that all vagueness must be eliminated; the rules must describe operations that are so simple and so well defined that they can be executed by a machine. Furthermore, an algorithm must always terminate after a finite number of steps.”<sup>7</sup> The actual application of this conceptualization into practical working systems took almost three hundred years: moving from the desire for a “machine computer” with Babbage in the early 19<sup>th</sup> century, to the application of computing to specific information processing activities under business machine manufacturers in the early 20<sup>th</sup>

---

<sup>3</sup> D. Knuth *Selected Papers on Computer Science* (Stanford, CA: CSLI Publications, 1996), chapters 0-1.

<sup>4</sup> D. Berlinski, *The Advent of the Algorithm: The Idea that Rules the World* (New York: Harcourt, Inc., 2000); M. Ridley, *Genome: The Autobiography of a Species in 23 Chapters* (New York: Perennial, 1999).

<sup>5</sup> Berlinski, *The Advent of the Algorithm*.

<sup>6</sup> Berlinski, *The Advent of the Algorithm*.

<sup>7</sup> Knuth *Selected Papers*, 59.

century, to the building of binary, coded general purpose mainframe machines in the 1960's, and finally to the extension of such systems through the rise of personal computing and internets.<sup>8</sup>

Overall, the technical patterns that structure information technologies currently can be summarized as follows<sup>9</sup>:

- 1) The expansion of binary logics and architectures to new knowledge-domains, giving rise to mathematically derived, digital mediums of information exchange.
- 2) The accelerating exploration and discovery of natural and social algorithms that can be translated to binary forms and architectures.
- 3) A constant miniaturization of processing power into smaller and smaller components.
- 4) The pervasive interconnectedness and interoperability of all technology systems (as digitally and algorithmically defined), increasingly wireless and embedded in both organic and inorganic models.
- 5) The increasing dominance of software over hardware as the optimal solution for understanding, developing and implementing digital, algorithmic processes.

The push for flexible, general “information machines” that characterized the historic development of computing has increasingly involved and been more easily achieved through software tools and methods. Not only is computation increasingly software derived or dependent, but the Internet itself — with algorithms embedded via software in TCP/IP, web browsers, network routing and management, Java scripts, databases and commerce to name only a few — is a reflection of the dominance of software as the central method for digitalizing information and implementing algorithmic-based processes.

### **III. Detailing Software Development: Process, Rationalization and Domain-knowledge**

The widespread existence of algorithms in social and natural processes creates a huge catalog of patterns that form a natural bridge between new technologies and society. This bridge is increasingly being crossed, and the lines being blurred, between what is beyond the reach of information technologies to understand, and thus transform and manipulate<sup>10</sup>. The means for bridging different domains of knowledge is occurring through software practices and products. In essence, if one aspect of the new global environment is new architectures of connectivity through new technologies, then software is the bricks and programmers the bricklayers of the networked society. As such, understanding how social processes are defined and transformed into digital logics — how informational architectures are defined and built — becomes essential to understanding the structures and impacts of new technologies in the economy and society.

---

<sup>8</sup> M. Campbell-Kelly and W. Aspray, *Computer: A History of the Information Machine* (New York: Basic Books, 1996).

<sup>9</sup> K. Eischen, *Information Technology: History, Practice and Implications for Development* (University of California, Santa Cruz, Center for Global, International and Regional Studies, Working Paper 2000-4, 2000) [www2.ucsc.edu/cgirs](http://www2.ucsc.edu/cgirs).

<sup>10</sup> R. Kurzweil *The Age of Spiritual Machines: When Computers Exceed Human Intelligence* (New York: Viking, 1999).

The technical aspects of software are not sufficient to understand this process. While the general algorithmic, digital patterns of software are constant, the technical patterns are built upon socially derived perceptions and understandings, not fixed universal, physical laws. Detailing the production of software places information, as opposed to technology, at the center of the analysis. This opens space to define the basic processes that structure the production and distribution of information in the global environment, clarifying the informational, qualitative and social aspects of new technologies like software.

### ***i. Software Development Processes: Producing Code***

If software is the brick and software programmers the bricklayers than software code is the raw material of information technology. Code is the basic language in which processes are translated into the binary code that can be processed by computers. The structure of creating software code follows a very basic pattern. Any process, such as typing words on a page or the interaction of two computers over the Internet, is defined in high-level language, that is on a macro-level of the basic operational procedures and outcomes. This is then detailed in a low-level design, a design that outlines the specific structure of the process. This design is then translated, or programmed, into the actual software language or code. All code does have rules of syntax and semantics, but there are multiple software languages and thus multiple forms in which to program low-level design. At either the low-level design or coding phase the actual process being modeled is placed into an algorithmic form, that is the logical mathematical sequence of events that will produce a desired outcome. Once placed in coded form, the program is then compiled, meaning translated from one of any number of programming languages that humans understand into “machine code” or the binary form composed of zeros and ones (e.g. 0010100011) that computers can understand. At this point, this is software and no longer code. It is an actual program that can be replicated and distributed as a complete product.

It is entirely possible, in fact almost guaranteed, that the software will not produce the desired results as specified in the initial high-level design — typing “A” may produce “a” or even worse “Z”, computers may be able to connect but unable to transfer data<sup>11</sup>. Such problems are commonly referred to as bugs, and the last stage of software development is removing these problems before the product is finally released. This involves translating (or un-compiling) software back into code to locate and remove the problem. Bugs, however, can stem from a number of problems along the whole development process, from basic flaws in the high-level design, mis-specifications in the low-level design, faulty algorithms, or simply mis-coding or even just mis-typing.<sup>12</sup> As such “debugging” can be a time consuming and demanding process that is a combination of luck, skill, experience and sheer effort. Once debugged, however, the software is then recompiled and finally complete.

The software development process, the human-aspect of producing code, mirrors these stages as well: high level design, low level design, code, unit test, system test, beta test (most recently), and final deployment.<sup>13</sup> Some aspects of the process are automated, but generally the process is

---

<sup>11</sup> N. Stephenson, *In the beginning ...was the command line* (New York: Avon Books, 1999)

<sup>12</sup> D. Kohanski, *The Philosophical Programmer: Reflections on the Moth in the Machine* (New York: St. Martin's Press, 1998).

<sup>13</sup> P. Jalote, *An Integrated Approach to Software Engineering* (New York: Springer, 1997)

human-centered. Standard software development processes entail the involvement of the same set of programmers at each of these stages for small projects. In other words, programmers conceive of the design, implement the code and debug the final product. As projects grow in size and complexity, the pattern is slightly altered. Project leaders determine high-level design in conjunction with end-users or project requirements, working out the overall system structure and some aspects of the detailed design specifications. The overall project is broken into localized “units” linked with programming teams that follow the same pattern as in standard, small project software development. Teams of programmers are responsible for coding the specifications (often creating the detailed algorithms that reflect the operationalizing of detailed specifications), testing the individual units and merging the units for an overall system test. Debugging, and even testing, is usually a constant process done internally by the programmers themselves.

The development process is structured, with aspects of automation and engineering (for example some analysis of efficiency or testing and locating bugs), but is fundamentally a “labor-intensive, intellectually complex, and costly activity in which good management and communication count for much more than technology”<sup>14</sup>. This structures software development around: 1) the need for skilled labor, and 2) the flows of communication throughout the production process. In the first aspect, programming — referring generally to the broad process of designing, coding and testing — is a labor-intensive process requiring specialized skills. Labor must be specifically trained in languages (e.g. HTML, Java, C++, Perl) and concepts of software development (e.g. software architectures, mathematical structures). Given these unique prerequisites, other engineering skills are not easily transferred to software production in time of increasing demand. Equally importantly, the skilled design and quality aspects of the process invariably require contingent decision-making and tacit knowledge that is generally learned “by doing”. Overall, the issue of labor is an issue of productivity, with skilled, experienced programmers in high demand to meet increasingly complex and new application demands.

In terms of the second aspect, increasing the complexity of software projects (usually in correlation with the process being modeled) corresponds to new demands for communication at all levels of the development process. As complex systems are broken up into manageable production units with greater numbers of programmers, the lines of communication between individual units and programmer increases dramatically<sup>15</sup>. This results in added levels of complexity as software is moved from design to final product. This is a fundamental aspect of software development. Larger, more complex software products require corresponding increases in the amount of direct labor needed for their production. Yet, the increasing number of individuals involved in a project actually decreases the efficiency of software production as the lines of communication increase, especially as programmers are brought in late in the process. Software development is at its most coherent when each of the development stages is combined within a single individual. As software projects increase in size and complexity, from individual to team to firm, the communications transaction costs increase significantly, often resulting in poor quality, missed shipping dates and cost-overruns.

---

<sup>14</sup> N. Fenton, S. L. Pfleeger and R. Glass (1994) “Science and Substance: A Challenge to Software Engineers” *IEEE Software*, Volume 11, Number 4, July (1994), 6.

<sup>15</sup> Specifically  $N(N-1)/2$  where N is the number of units and/or programmers. See F. Brooks, *The Mythical Man-month: Essays on Software Engineering* (Reading, MA: Addison-Wesley, 1995), 18-19.



## ii. **Rationalization Software Development**

The evolution of software development over the last thirty years — driven by increasing demand for new and more complex software— has been consistently structured by the dynamic interaction of the need for high quality labor and communication limits at each stage of software production. Generally, software development has focused on increased productivity through skilled-labor, minimizing issues of quality and complexity arising from issues of communication. Most software development models deal with the increasing demands made on software production through an emphasis on extensive growth, constantly meeting new project needs through ever more skilled-labor.<sup>16</sup>

Importantly, even the most celebrated success stories, specifically the open-source movement and Microsoft, rely on large amounts of unpaid labor to overcome the fundamental inefficiencies and bottlenecks of the software process itself.<sup>17</sup> The open source movement — most well known through the Linux OS, Apache and SendMail programs — relies on distributed networks of programmers who write and debug code based on ties of community and ideology and not financial compensation.<sup>18</sup> Software development has also moved to the standard practice of beta-testing, the shipping of trial versions of software to groups of users, usually at no cost, prior to final product shipment. One of Microsoft's chief strengths, and one that compensates for the firm's weaknesses in the software development process, is its extensive network of beta-testers. Windows 95 was shipped to 400,000 beta-testers prior to final release. Such beta-testing has been known to detect 25% of all bugs in a system.<sup>19</sup>

However, such extensive solutions have not resulted in either relative increases in productivity or quality given the increases in complexity of software products. The emphasis on labor has not minimized the essential problem of communication and its impact on quality as projects expand in size. Software is expensive to develop, usually late, often unreliable and subject to constant change and rework. Up to 80% of software projects are never completed<sup>20</sup>, but remain half finished, never moving beyond abstract design. By the mid-1980's, software was considered to be 80% of the cost of new computer systems<sup>21</sup>. Seventy-five percent of that cost was not development but maintenance (debugging, upgrades and integration with other software products) of the software after installation<sup>22</sup>.

This overall pattern has raised the issue of process rationalization as a central question within software development. The basic concerns of increasing product complexity, low quality and weak productivity growth framed by a skilled-labor and highly interactive process push for

---

<sup>16</sup> Brooks, *The Mythical Man-month*; M.A. Cusumano and R.W. Selby, *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People* (New York: The Free Press, 1995); E. Raymond, Eric (1999) *The Cathedral and the Bazaar: Musings on Linux and Open Source By an Accidental Revolutionary* (Sebastopol, California: O'Reilly, 1999).

<sup>17</sup> Cusumano and Selby, *Microsoft Secrets*.

<sup>18</sup> Raymond, *The Cathedral and the Bazaar*.

<sup>19</sup> Cusumano and Selby, *Microsoft Secrets*, 309-313.

<sup>20</sup> Hoch et al, *Secrets of Software Success*.

<sup>21</sup> OECD, *The Software Sector: A Statistical Profile for Selected OECD Countries* (Paris: Committee for Information, Computer and Communications Policy, January, 1998) <http://www.oecd.org>

<sup>22</sup> Jalote, *An Integrated Approach to Software Engineering*, 3.

rationalizing the development process. Defining or automating software development, like any industrial process, would overcome reliance on skilled-labor, raise productivity and quality, and place new product development within structured, managed practices. Yet, software development has continually resisted the creation of a repeatable, quantifiable and improvable process. A series of “magic bullets” — ranging from new development methodologies to new programming languages — have been presented that have promised to radically alter the nature of production, once and for all moving software development towards a rational process. None have succeeded.<sup>23</sup> The process has remained manageable, but not rationalized or systemized industry-wide. New tools and methods have failed to relieve the constant labor shortage, software quality and productivity continues to be unpredictable and unquantifiable, and new defined organizational methods have failed to overcome the limits of communication. Answering why software resists rationalization is at the heart of understanding software itself.

### a. *A Brief Outline of Software Development Methods*

The challenges of software development are well-recognized within the software community. Solutions, however, remain unclear. Within the software profession, there has been a constant debate on the nature of software development fluctuating between engineering and craft approaches for almost thirty years. Many view software — focusing on its digital, algorithmic and Computer Science foundations — as a discipline capable of being engineered through the use of scientific and mathematical approaches as given in the dictionary definition above. Alternative approaches view software — focusing on the domain-knowledge translating, tacit, human centered aspects — as an almost artistic process, describing it as a craft produced by “softcrafters”<sup>24</sup> or craftsmen.<sup>25</sup> Looking at the history of the debate, there are clear cycles of alternating strength between engineering and craft 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.<sup>26</sup>

---

<sup>23</sup> F. Brooks, “No Silver Bullet: Essence and Accidents of Software Engineering” *Computer*, April (1987). Structured design and programming have been attempted. Process models like the Waterfall (R. S. Pressman, “Software Engineering”, in M. Dorman and R.H. Thayer, Editors, *Software Engineering: A Practitioner’s Approach* (New York: McGraw-Hill, 1997), Spiral (B. Boehm, “A Spiral Model of Software Development and Enhancement”, *Computer*, May (1998)), Software Life-Cycle (A.M. Davis, “Software Life Cycle Models”, in R. Thayer, Editor, *Software Engineering Project Management*, Los Alamitos, CA: IEEE Computer Society, 1997), personal process model (P. Ferguson, W. Humphrey, S. Khajenoori, S. Macke and A. Matvya, “Results of Applying the Personal Software Process” *Computer* May, (1997)) and open source (Raymond, *The Cathedral and the Bazaar*) have been implemented. New programming languages such as C++ that are based in object oriented programming have been used to maximize software reuse and maintenance. Software metrics have been created that attempt to quantify and map the software process based on TQM principles (R.B. Grady, “Successfully Applying Software Metrics”, *Computer*, Volume 27, Number 9, September, 1994). Cost and quality models such as COCOMO (D.B. Legg, “Synopsis of COCOMO” in Thayer, *Software Engineering Project Management*,) and the Capability Maturity Model (J. Herbsleb, D. Zubrow, D. Goldenson, W. Hayes “Software Quality and the Capability Maturity Model” *Communications of the ACM*, Volume 40, Number 6, June (1997)) have been developed.

<sup>24</sup> T. Gilb, *What is Level 6* (1996). <http://stsc.hill.af.mil/swtesting/gilb.asp>

<sup>25</sup> P. McBreen, *Software Craftsmanship: the New Imperative* (New York: Addison-Wesley, 2002).

<sup>26</sup> 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.

The debate itself has existed from the beginning of software as an independent activity within computer development. The most significant and referenced event in software's establishment as an independent activity was a 1968 NATO conference. The meeting gathered to consider general concerns and management issues linked to a pending "software crisis". The "crisis" being the increasing demand for software, in an environment of limited skilled-labor and low software quality. The conference produced widespread agreement around the clear need to prevent the crisis through the development of "software engineering"<sup>27</sup>. "The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering."<sup>28</sup>

More interesting, and much less well considered, is the follow-up 1969 conference and report. The second meeting was designed as an integral follow-up to the first, intended to move from general concerns of software development to specific technical questions and methods of preventing the "crisis". In other words, the second conference was intended to generate consensus on the form and practice of 'software engineering'. However, the gathering failed to generate such consensus. The meeting came to be dominated by a "communication gap" between participants, particularly between theoretical and practical positions on software development.<sup>29</sup> Theoretical approaches, usually affiliated with Computer Science views, emphasized hypothetical and mathematical but unproven techniques. Practical approaches, labeled software engineering, emphasized results in real world settings even if using less than optimal techniques.

The two conferences demonstrate that, from its conception as an independent field of activity, software development has faced serious and unresolved tensions between issues of management, theory and practice. While the terms have changed — engineering having become the established theoretical model in contrast to real-world craft approaches, the fundamental conflicts and debate still address the same issues.<sup>30</sup> There is generally agreement that software should be produced at the highest quality for the lowest cost (hence, the agreement at the first conference addressing general issues and management questions). 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 between theory and practice).

### *b. Limits to Rationalizing Software Development*

Even with the outline of software development above, it remains unclear why quantifying software practices remains so difficult. Historically, 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

---

<sup>27</sup> *Software Engineering: Report on a Conference sponsored by the NATO Science Committee*, Garmisch, Germany, 7<sup>th</sup>-11<sup>th</sup> October, Peter Naur and Brian Randell, Editors (1968).

<sup>28</sup> *Software Engineering: Report on a Conference*: 8.

<sup>29</sup> *Software Engineering Techniques: Report on a conference sponsored by the NATO Science Committee* Rome, Italy, 27<sup>th</sup> -31<sup>st</sup> October, J.N. Buxton and B. Randell, Editors (1970).

<sup>30</sup> A. Cockburn and J. Highsmith "Agile Software Development: The Business of Innovation" *Computer*, September (2001), 120-122; S. R. Rakinin, Letters, *Computer*, December (2001); B. Boehm "Get Ready for Agile Methods, with Care" *Computer*, January (2002): 64-69.

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 of a craft who were not involved in nor inclined to build software factories or engineered processes. However, this still leaves open to explanation why the debate and central issues have continued almost unchanged to the present.

The history of industrial society is full of conflicts and debates between craft and engineered approaches. 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 was, and 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. 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 is not sufficient in and of itself to prevent an overall transformation of the industry towards engineered methods.

So, 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 manufacturing”, “software engineering”, defined divisions of labor, economies of scale—become dominant within the industry?<sup>31</sup> The simple reason is communication<sup>32</sup>. This is not new in and of itself. 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<sup>33</sup>, both within the development process and its translation function. As early as 1974, Brooks detailed the specific limits of software development as not the number or type of tools or a lack of specific organizational structure, but rather the increase in lines and quality of communication. This is important and usually misunderstood. The mathematical nature of software would seem to indicate that the communicative aspects would be rigorously defined, both in establishing the basic requirements and in the coding of software. Software approaches and thinking are not only distinct from math<sup>34</sup>, but are not mathematically derived any more than music is purely notes written on a page.<sup>35</sup> It is the process of designing the algorithmic patterns and linking such

---

<sup>31</sup> C.f. M. Cusumano, *Japan's Software Factories: A Challenge to U.S. Management*, New York: Oxford University Press, 1991); H. Baetjer, *Software as Capital: an Economic Perspective on Software Engineering* (Los Alamitos, CA: IEEE Computer Society, 1998); M. Poppendieck, “Lean Programming” *Software Development*, May (2001).

<sup>32</sup> McBreen, *Software Craftsmanship*.

<sup>33</sup> P. G. Armour, “The Case for a New Business Model”, *Communications of the ACM*, Volume 43, Number 8, August (2000).

<sup>34</sup> Knuth, *Selected Papers*, chapter 5.

<sup>35</sup> Gelernter, *Mirror Worlds*, 38-41.

abstractions to concrete results throughout the development process that limits quantification and rationalization.

Software development, and building basic requirements, is a process of communicating tacit knowledge.<sup>36</sup> 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. Mapping the algorithmic form of a process involves subjective criteria, even of quality and efficiency, based on perceptions of the process being modeled and expected outcomes.

“A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct. Programs often need to be modified, because requirements and equipment change. Programs often need to be combined with other programs. Success at these endeavors is directly linked to the effectiveness of a programmer’s expository skills.”<sup>37</sup>

Defining quality and efficiency requires measuring differences in logical approach to the process being modeled that involve issues of style, aesthetics, robustness, integration, evolution and cost. Even the choice of programming languages doesn’t rigorously define outcomes. Programmers do use rules of syntax and semantics, but like all languages both the structure and the meaning vary over time and space, even between programmers within a project. There is no automatic translation of designs or requirements into a programming language. Like all languages, there are multiple ways to communicate similar concepts and outcomes. This interpretive, craft process means that there are no universal physical laws to define all software development, and thus no direct method to quantify software processes and products (and thus people) against a universal standard of quality and efficiency. This informational, communicative aspect of software development inherently limits its rationalization, and thus its engineerability and industrialization.

This helps define the importance of communication in software development. Even between programmers, communicating tacit-knowledge or organizing knowledge flows, presents a serious challenge. This explains why software development is most successful when encapsulated in one individual. Equally significant, the need for communication is broader than merely having adequate documentation, telecommunications or organizational structures. The mapping of real world processes into algorithmic patterns that contain multiple contingencies is fundamentally structured by the assumptions, domain-knowledge and communication of designers, programmers and end-users. The communication between end-users and programmers must be both highly detailed and interactive at all levels of the production process in order for specific and often tacit domain-knowledge to be successfully translated into a functioning product. Quantifying or rationalizing such communication, between projects or teams or firms, is exceedingly difficult and fraught with subjectivity.

---

<sup>36</sup> P.Armour “The Spiritual Life of Projects: the Human Factor in Software Development is the Ingredient that Ultimately Gives a Project Team its Soul” *Communications of the ACM* v45, n1 (Jan, 2002): 11.

<sup>37</sup> Knuth, *Selected Papers*, 2.

Design in any industry is always a challenging intellectual activity. It is even more so in software, exactly because it is essentially a pure design process. This debate surrounding development methods is a reflection of this fundamental characteristic. Software engineering methods come out of a government and military tradition where problems and needs are believed to be clearly defined. Issues of management, cost and robustness, not how to define the problem domain, are central. Craft methods come out of university and programmer communities, where solutions to problems evolve overtime, collectively, based on an open-ended transparent process. Both methods address the specific demands of translating domain-knowledge, though they make fundamentally different assumptions on the definability, review and evolution of such knowledge.

The human-centered, communicative aspects of software processes directly limit quantifying time, cost, quality or productivity systematically and consistently over time and space<sup>38</sup>. A rationalization of the development process would, by definition, create a defined division of labor that would stratify development into skilled and unskilled aspects. Developing such a division of labor involves defining software requirements in universal, fixed formats, eliminating the need for interactive, craft development methods. The domain-knowledge and design basis of software preclude this from happening. Social scientists should clearly understand the difficulty in defining and quantifying, particularly universally and over time, patterns that involves human interaction, practice or belief. Yet, this is exactly what is required for software development to be rationalized.

General patterns can be recognized, but as software moves from higher to lower levels of abstraction, general patterns increasingly become specific differences open to interpretation. Especially when mapping social process, much of the domain-knowledge is tacit, undefined, uncodified, developed over time, and often not even explicit to the individuals actively involved in the social process. Yet, software demands a defined process through its binary, algorithmic structure, even if none apparently exists or is understood internally or externally. Even where such patterns are assumed to be rigorously and fully mathematically defined, there still remains the strong possibility of randomness or incompleteness in the algorithms themselves.<sup>39</sup> Equally important, such social knowledge and practices are dynamic, constantly evolving and transforming. It should not be surprising that rationalizing the modeling of such processes is exceedingly difficult, and that such efforts are often incomplete, impractical or unsatisfactory.

### ***iii. The Central Aspect of Domain-knowledge: From Algorithms to Social Architectures***

The translation of domain-knowledge into codified algorithms clarifies how software development accesses and transforms information.<sup>40</sup> This structures software as a design driven, socially-based practice. Software, because of its flexibility, adaptability and role in structuring the protocols and content of the Internet, is increasingly the chosen form for the application of coded knowledge in society. The translation aspect signals that software is essentially embodied social knowledge, creating digital and algorithmic models of social processes directly or

---

<sup>38</sup> W. Gibbs, "Software's Chronic Crisis", *Scientific American*, Volume 271, Number 3, September, (1994).

<sup>39</sup> T. Norretranders *The User Illusion: Cutting Consciousness Down to Size* (New York: Viking, 1998): chapter 3

<sup>40</sup> Mitchell, *City of Bits*; Negroponte, *Being Digital*.

modeling processes filtered through limited domain-knowledge. Fundamentally, how easily a concept is translated is a function of the nature of domain-knowledge itself in society. The farther away from broadly accepted and understood domain-knowledge the more difficult a concept is to translate, and it is clear that most domain-knowledge is context specific, changing with culture, location, gender and experience to name just a few possibilities.

The greater the translation from analog to digital procedures, the greater the knowledge embedded in software and information technology. In this way, it has been argued that software is not a product to be produced, but a process of storing knowledge. Software – like DNA, the human brain, hardware and books – is just the most recent medium for knowledge storage.<sup>41</sup> However, the unique feature of software, as compared to these other mediums of knowledge storage and communication is its persistency, update speed, deliberate flexibility, and applicability to action.<sup>42</sup> Software has each of these qualities that are either absent or less dynamic in the case of the other mediums. More importantly, and perhaps the greatest strength of software, is its ability to emulate each of these other mediums of knowledge. For example, software can emulate the best features of books, and is increasingly replacing hardware, especially at the integrated circuit level, in many cases.<sup>43</sup> It is this basic structure and scope that gives software its unique place among new technologies and within the information society.

The social impact of this is quite direct. Software is an incredibly flexible and adaptable medium of knowledge storage and communication, making it an increasingly essential feature of society. However, the very process of translating knowledge into software, because of the specific demands of the algorithmic patterns underlying its coded format, requires that domain-knowledge be rigorously and unambiguously defined. The process of translating this knowledge and designing the coded, algorithmic formats tends to be situational and crafted, rather than defined and manufactured. The end result being that the defined, algorithmic patterns that structure knowledge and communication are built through opaque, human-centered practices that reflect specific social, organizational and individual norms, beliefs and priorities.

Though software operates within a veneer of engineering or mathematics, the fundamental reality is that software's structure and impact is socially determined. Software developed by two different programmers or organizations, within different development environments, from distinct cultures or experiences, possessing distinct domain-knowledge will result in distinct software. This fact is reflected in the debates surrounding open-source and proprietary software development that involve two distinct visions of how software should be made, distributed and rewarded simultaneously. The explicit issue is that the nature of software development reflects directly on the quality, social applicability, cost and innovation of software products. The importance of the debate is that unlike standard commodities, software as embedded domain-knowledge with a tendency towards a network effect can establish default patterns of communication and knowledge that directly structure social interactions, organizations and institutions.

---

<sup>41</sup> Armour, "The Case for a New Business Model", *Communications of the ACM*, Volume 43, Number 8, August 2000.

<sup>42</sup> P. G. Armour, *The Case for a New Business Model*.

<sup>43</sup> The case of Transmeta, which has created a software structured IC designed specifically for mobile application, is an ideal example of this (*Wired* July 2000).

Overall, software processes and architectures will increasingly influence the development and organization of many of the basic social structures being constructed within an information dominated society as they become codified into digital forms. This is exactly why “code is law,”<sup>44</sup> that is a fundamental pattern that structures spatial and temporal, and thus social and economic, interactions. The basic development structure, combined with the congealed knowledge inherent in software, distinctly influence the development of the information technology industry, which in turn is increasingly influencing industry globalization patterns, debates over intellectual property, proprietary versus open-source production models, basic public safety issues, and economic development strategies.<sup>45</sup> Understanding software as both embodied social knowledge<sup>46</sup> and as a fundamental production process within society, directly helps frame an understanding of what the social and economic implications of information technologies are.

#### **IV. The Social Impact of Software Practice: Failures, Absence and Bounded Behaviors**

The communicative, knowledge-driven aspects of software development explain many of the ramifications of software in society. It also helps to understand the limitations of how such impacts are conceptualized and debated. First, because of the domain-knowledge basis of software production, software products are inherently limited, even flawed.<sup>47</sup> Even beyond the potential for variation or error within a non-rational process, the reliability and impact of software products is structured both through its social, human-centered development process and its defined algorithmic nature. Software products inherently contain limitations based in the assumptions used to place analog, continuous processes within digital, discrete patterns. Common examples of this, from the Y2K problem (which assumed a pace and direction of computer advancement) to noted failures at NASA (where human discrepancies on choice of measurement systems have caused system failures), clearly demonstrate the limits of software to offer robust (five “nines” reliability) products as compared with mechanical systems. Other failures of large systems, including well-publicized failures at both the Hong Kong and Denver airports<sup>48</sup>, demonstrate the limits of mapping all possible contingencies of complex systems within software’s algorithmic structures. Such concerns become more important when considering the emulation of DNA or ecosystems or energy markets

More importantly, however, even “reliable” software structures interactions within the framework of the assumptions built into the product. Trivial examples abound from the need to go to the “Start” menu in Windows to shut down a computer to the very concept of a “desktop” as a metaphor for interaction with computing systems.<sup>49</sup> An even more important example, given the two billion illiterate individuals in the world, is the assumption that interactions with digital

---

<sup>44</sup> Mitchell, *City of Bits*.

<sup>45</sup> Lessig, *Code*.

<sup>46</sup> Agre and Schuler, *Reinventing Technology, Rediscovering Community*

<sup>47</sup> L. Wiener, *Digital Woes: Why we should not Depend on Software* (Reading, Mass.: Addison-Wesley Pub. Co., 1993).

<sup>48</sup> E. Ullman, “The Myth of Order”, *Wired*, Version 7.04, April (1999): <http://www.wired.com>

<sup>49</sup> M. Dertouzos, *The Unfinished Revolution: Human-centered Computers and What They Can Do for Us* (New York: HarperCollins, 2001).



technologies should be text based and in Roman characters.<sup>50</sup> The social impacts of the assumptions built into digital architectures and environments are far more important, but recognized and debated far less than the software failures above. Discussions of the “Digital Divide” or “access to information” or a whole host of other social issues take on complete new perspectives when thought of in terms of the social assumptions embedded in software products.

A specific and illustrative example of this is that of Bangla, the dominant language in Bangladesh and West Bengal and the eighth largest linguistic group in the world, that is unrepresentable in any of the major software operating systems in the world. The lack of representation is a result of the lack of enforcement of copyright laws within Bangladesh that has removed incentives by companies to invest in Bangla language versions of software products, as well as the countries failure to pay the US\$12,000 to join the international Unicode Consortium standard that would insure a standard linguistic code for the software interface. The problem extends further when large tribal populations use the Bangla alphabet in their languages.<sup>51</sup> So in this case, the “Digital Divide” is not only or even about access to hardware or software products or “informational content”, but begins with the formulation of software architectures, the financial and political resources to participate in global standards settings about these architectures, and the acceptance of global industry views of property rights for digital products.

The impact of software is not necessarily always direct, but no less potentially important. Two other interesting, if only suggestive, examples of the impact of software on modern accounting. On one hand the market domination of specific software products for all of the leading accounting firms limits both the differentiation between the firms, as well as structures their strategic evolution as they seek to adapt these basic packages to their individual firm needs.<sup>52</sup> Reliance on specific software causes firms to accept the basic accounting structures and practices inherent built into the software. Additionally, it has been suggested that the recent accounting scandal and collapse of Enron can be blamed on Microsoft’s dominance in software, where the predominance of the desktop metaphor creates opaqueness in the very transactions that regulators need to monitor.<sup>53</sup> The suggestion here is that both viewpoints indicate that software products structure the practices of users of software, with social consequences beyond whether such software is reliable or economically valuable under traditional conceptualizations.

Each of these examples, while clearly cursory, suggest the importance of considering how the software’s information practice frames scientific and social debates and impacts.<sup>54</sup> The technical and social aspects of software development structure software products as embodying norms and regulations<sup>55</sup> that regulate social interactions in a *defacto* manner. The increasing role of government, particularly in the United States, in protecting or promoting such monopolies

---

<sup>50</sup> Dertouzos, *The Unfinished Revolution*.

<sup>51</sup> M. Hossain “Bangla Still Off Computer Code” *The Daily Star*, February 21<sup>st</sup>, 2002.

<sup>52</sup> M. Dirsmith, M. Fischer and S. Samuel *Technology and the Changing Location of Expertise in Knowledge-Based Professional Organizations*. Presented at the ASA Annual Meeting, Anaheim, CA., August 19<sup>th</sup> 2001.

<sup>53</sup> K. Maney *Scientist says its Time to Change ‘Virtual Tupperware’ System* USA Today, February 20<sup>th</sup>, 2002, 3b

<sup>54</sup> As articulated by Bill Joy and Amory Lovins in *Wired*, August (2000).

<sup>55</sup> Lessig, *Code*.

without understanding their full impact both locally and globally is of increasing importance.<sup>56</sup> The legal and institutional frameworks established around software products will shape in large measure the durability, control and openness of new patterns of power and profit in an information society.

## V. In Conclusion: Opening Software Practice to Society

Kranzberg's Law states "technology is neither good, nor bad nor neutral,"<sup>57</sup> meaning that society determines the application and impact of technology after its initial development. However, the case of informational technologies, and software in particular, help elaborate how technological architectures and development processes define frameworks for future social action even prior to their application in society. Software is a way in which knowledge or information is translated to a digital form, as such it embodies social knowledge. This means that prior to its application by society, software is already replicating and structuring the forms that social interactions may take. Combined with Metcalfe's Law<sup>58</sup>, this implies that once certain digital patterns of interaction become dominant – such as the protocols that shape Internet communication or define how computers structure information – the social interpretation or regulation of these technologies is already limited.

The partial weakness of information society theories has been the failure to recognize and unpack the central process of domain knowledge transformation at the micro-level. The informational aspect of new technologies does not signal only communication, but the embodying of social practices and interactions in technology itself.<sup>59</sup> Software development is the process of converting social knowledges and practices into digital form, so that they can be manipulated, disseminated and controlled within a coded binary architecture. This process of translation is really a process of transformation. Norms and values are implicitly included in the algorithms that shape the coding of the social processes represented by particular domain-knowledges.<sup>60</sup> Informational practices like software development increasingly shape the rules and procedures through which social interaction takes place. Code, in whatever digital or algorithmic form, is thus a new form of law, one that is flexible, but also inherently social.<sup>61</sup>

Mapping how social practices and knowledge are codified into software is a central aspect to understanding the broader patterns structuring social interaction in an informational environment.<sup>62</sup> Software, or information technologies generally, is not only a physical

---

<sup>56</sup> J. Hibbard "The Open-source Debate enters the Genomics Arena: Should Publicly Funded Software be Free?" (*Red Herring*, February 25<sup>th</sup>, 2002); O. Malik "A Y2K Bug for Health Care" (*Red Herring*, February 27<sup>th</sup>, 2002); B. King "Can the World Be Copyrighted" (*Wired News*, February 26<sup>th</sup>, 2002).

<sup>57</sup> M. Kranzberg "The Information Age: Evolution or Revolution?" in B. R. Guile, Editor, *Information Technologies and Social Transformation*, (Washington D.C.: National Academy Press, 1985).

<sup>58</sup> Metcalfe's Law states that the "value" or "power" of a network increases in proportion to roughly the square (for large numbers) of the number of nodes on the network, or more precisely  $N^2-N$ .

<sup>59</sup> P. Agre and Do. Schuler, Editors, *Reinventing Technology, Rediscovering Community: Critical Explorations of Computing as a Social Practice* (Palo Alto: Ablex, 1997).

<sup>60</sup> H. Nissenbaum "How Computer Systems Embody Values" *Computer*, March (2001); P. Agre "Toward a Critical Technical Practice: Lessons Learned in Trying to Reform AI" in G. Bowker, L. Gasser, L. Star and B. Turner, Editors, *Bridging the Great Divide: Social Science, Technical Systems and Cooperative Work* (Erlbaum, 1997).

<sup>61</sup> Lessig, *Code*.

<sup>62</sup> P. Agre "The Market Logic of Information" *Knowledge, Technology, and Policy* 13(1), (2001) 67-77.

architecture that facilitates global flows<sup>63</sup>, but also structures how such flows occur and their impact.<sup>64</sup> Recent studies have attempted to clarify the “laws of the web”. Using mathematical techniques derived from mapping large distributed systems, the research seeks to demonstrate that patterns do emerge from apparently chaotic random systems like the World Wide Web.<sup>65</sup> The central assumption is that the Web is a random system to begin with. But such an assumption can only be made when there is no real discussion or understanding of the socially derived standards and tools — essentially software in its algorithmic digital form — that comprise the Web’s architecture. In other words, the architecture of the Web, like all architectures,<sup>66</sup> bounds possible behavior, and it is not surprising that the architecture produces consistent behavior, even in very large systems.<sup>67</sup>

The essential aspect is that the “laws of the web” are not simply outside of it, embedded in human behavior or the informational ecosystem and played out through the technology of the web, but in the software itself. How these new “mirror worlds”, as Gelernter calls these software constructed architectures and environments,<sup>68</sup> are formed is an essential question to understanding their ultimate characteristics and social impacts. Detailing software development helps map the process on which such frameworks are built, signaling the assumptions about social processes inherently built into the algorithms that comprise digital architectures. As software increases in importance in society, such hidden assumptions will increasingly impact upon how social relations are structured.

These new social architectures are an increasingly significant factor in the strategic and organizational choices of governments, firms and civil society, whether conscious or not<sup>69</sup>. Exactly because of this, debates on the nature of software development — particularly open-source and proprietary debates both within the profession and in their broader form in society — deserve special and broad attention. Ignoring such patterns limits our understanding and analysis of the broader social transformations in the global environment. More importantly, failing to recognize these processes limits the space for social debate, policy and action around the establishment and evolution of these new digital architectures at the locus of their development.

---

<sup>63</sup> Castells, *The Information Age*; Held et al, *Global Transformations*.

<sup>64</sup> J.S. Brown and P. Duguid, *The Social Life of Information* (Boston, Massachusetts: Harvard Business School Press, 2000); William Mitchell, *e-topia* (Cambridge: MIT Press, 2000).

<sup>65</sup> B. Huberman, *The Laws of the Web: Patterns in the Ecology of Information* (Cambridge, MA: MIT Press, 2001).

<sup>66</sup> For a theoretical consideration of the interaction of architecture and social practice see P. Hall *Cities of Tomorrow* (Oxford: Blackwell Publishers, 1996), and P. Bourdieu *Outline of a Theory of Practice* (New York: Cambridge University Press, 1977), particularly chapter 2 and pages 87 – 95.

<sup>67</sup> H. Simon *Models of Bounded Rationality: Empirically Grounded Economic Reason* (Cambridge, MA: MIT Press, 1982). It is interesting and arguably not surprising that Herbert Simon occupied a leading position as an economist and psychologist, and one of the leading founders of computer science.

<sup>68</sup> D. Gelernter, *Mirror worlds, or, The Day Software puts the Universe in a Shoebox--: How it will happen and What it will mean* (New York: Oxford University Press, 1991).

<sup>69</sup> Lessig, *Code*; W.J. Mitchell, *City of Bits: Space, Place and the Infobahn* (Cambridge: MIT Press, 1998).