Connections between the Software Crisis and Object-Oriented Programming

Hansen Hsu
Cornell University, Science & Technology Studies
SIGCIS History of Computing Workshop in Memory of Michael S. Mahoney

"It is the history of computer software, and not of the computer itself, that is at the heart of the larger story of the great 'computer revolution' of the mid-to-late 20th century."[1] "The history of software is the history of how various communities of practitioners have put their portion of the world into the computer… translating their experience and understanding of the world into computational models… [which] reflect the histories of the communities that created them and cannot be understood without knowledge of those histories, which extend beyond computers and computing to encompass the full range of human activities."[2] These two quotes, the first by Nathan Ensmenger in his upcoming book, and the second, opening his piece on "What Makes the History of Software Hard" by the late Michael Mahoney, when taken together outline a specific approach to the history of computing: software is treated as central, not an afterthought, and social relationships, not technological artifacts, are fore-grounded. This program brings the History of Computing and of software more closely in line with

[1] Nathan L. Ensmenger, *The "Computer Boys" Take Over: Computers, Programmers, and the Politics of Technical Expertise* (Cambridge, MA: MIT Press), Forthcoming, 5.
[2] Michael S. Mahoney, "What Makes the History of Software Hard," *Annals of the History of Computing, IEEE* 30, no. 3 (2008): 8.

History of Technology and with Science and Technology Studies, an agenda Mahoney famously put forward in 1988.[3]

This paper originally began as an exploration into the origins of a computing technology and system of practice, known as "object-oriented programming" (OOP), and upon a suggestion by my advisor, Ron Kline, into its relation to the "software crisis" of the 1960s, I began with an examination of Mahoney's 2002 paper on the software crisis, "Software, The Self-Programming Machine."[4] The work of Nathan Ensmenger at the University of Pennsylvania touches extensively on the software crisis, and his work builds on Mahoney's foundations. Both Mahoney and Ensmenger cast the software crisis as a traditional struggle between management and software workers, with both new management technologies and new software technologies (software engineering, structured programming, object-oriented programming) devised as solutions, primarily pushed by management, to solve the perceived crisis. Yet, many proponents of object-oriented programming are programmers, including computer science professors, who have hailed it for its technical advantages over previous methods, even claiming it to be a "revolution"[5] and a "silver bullet" that could solve the software crisis.[6] If object-oriented

[3] Michael S. Mahoney, "The History of Computing in the History of Technology," *Annals of the History of Computing, IEEE* 10, no. 2 (1988): 113-125, doi:10.1109/MAHC.1988.10011
[4] Michael S. Mahoney, "Software: The Self-Programming Machine," in *From 0 to 1: An Authoritative History of Modern Computing*, ed. Atsushi Akera and Frederik Nebeker (New York, N.Y: Oxford University Press, 2002), 91-100
[5] "There is a revolution brewing in the software community. Building software quickly, correctly, and economically remains an elusive goal, and this at a time when demands for new and more powerful software are soaring." Harvey M. Deitel and Paul J. Deitel, *C++ : how to program* (Englewood Cliffs N.J.: Prentice Hall, 1994), 10.
[6] Brad J. Cox, "There Is a Silver Bullet: A software industrial revolution based on reusable and interchangeable parts will alter the software universe," *BYTE*, October 1, 1990

programming originated solely out of managerial attempts to discipline software workers, how would one explain the enthusiasm of the programming community itself for it? Upon examination of the history of two important object-oriented languages, Smalltalk and C++, through the eyes of its creators, I find that the story is more complex: Smalltalk, the first object-oriented language, originated in a research environment and was more influenced by academic computer science. C++ also originated in a research environment but one more attuned to practical software industry concerns, resulting in a language that violated what some proponents believed to be core principles of object-orientation[7] in favor of closer similarity to older structural and procedural programming, yet in spite of, but possibly precisely because of this, becoming the most widely used object-oriented programming language in the industry,[8] thus making it the most likely OO language of choice for managers.

No "revolution" is without resistance, however, and though it is now almost two decades since that revolution supposedly began, and object-orientation is supposedly everywhere, its full benefits, as touted by its most enthusiastic proponents, have hardly been realized. The controversy over C++'s technical merits as an object-oriented language, especially its lack of a feature known as "dynamic typing and binding" common to other object-oriented languages, reveals a persistent tension in the history of

[7] "C++ is a significant barrier to learning and correctly applying object-oriented methodologies and technologies." Bruce Webster, *Pitfalls of object-oriented development* (New York: M&T Books, 1995), 139

[8] "the truth is that C++ is the closest thing to a standard among the object-oriented programming languages (OOPLs) in the world today. It is the most broadly supported, has the greatest variety of compilers and supporting tools, and is considered the standard development language for SunSoft's DOE, OS/2, the OpenDoc initiative, Taligent, [the last two both collaborative projects involving Apple in the early-1990s] and Windows." Ibid., 140

software. The design of programming languages over time has tended to proceed in the direction of higher levels of abstraction, partly in an effort to manage the complexity seen to be at the root of the software crisis. Yet as seen in C++, alternate values, practicality and efficiency, are often invoked to justify departures from high level abstraction, giving programmers low level access to hardware. Bruce Webster, in his book, *Pitfalls of Object-Oriented Development*, implies that widespread use of C++ has actually hampered the spread of good object-oriented practice.[9] A recurring theme in his book is that proper object-oriented design methodology cannot be imposed by the technology (the language) itself, but that the designer must first transform his/her way of thinking over to the new "paradigm."[10] I examine the controversy over dynamic typing and binding in light of the histories of Smalltalk and C++ as a possible counterexample to the thesis that management favors higher abstractions to manage complexity.

Mahoney and Ensmenger's Theses Considered

The canonical story of the software crisis proceeds thus: the mid-to-late 1960s marked a period where increases in hardware capability appeared to lead to an increase in the complexity of computer programs. Programming had, very early on, been considered to be, in the words of John Backus, inventor of the high-level programming language

[9] Ibid., 138-9, see also the quote in note 7 above.
[10] Proponents of object-oriented programming often describe it in Kuhn's language of paradigm shift. Brad Cox explicitly references Kuhn, while Bruce Webster gives a re-definition of "paradigm" in the context of software: "A particular approach or concept, used especially in software to refer to the way a given task is presented to and handled by the user. Brad J. Cox, "Planning the Software Industrial Revolution," *IEEE Software* 7, no. 6 (November): 25; Webster, *Pitfalls of object-oriented development*, 26, 126.

Fortran, a "black art,"[11] in the sense that it was a craft, artisanal skill requiring intuition, creativity, rules of thumb and tinkering. This was especially apparent in the early days of programming on hardware with limited capabilities, so that whatever tricks could be done to wring the maximum performance out of a program or minimize the amount of memory used would be employed. Better hardware freed software programmers to construct more capable, and thus more complex, software systems, which quickly grew too large for any one programmer to handle. The complexities of managing large groups of programmers led to widely publicized failures: IBM's OS/360 project cost the company half a billion dollars, and the Mariner I spacecraft veered off course and had to be destroyed due to a simple transcription error.[12] With hardware costs decreasing, the cost of creating and maintaining software, and therefore of programming labor, soon became the largest source of expenditure on computing. This situation gradually became known in the computer industry as the "software crisis." In response, Fred Brooks, the IBM programming manager who led the ill-fated OS/360 project, wrote the famous book, *The Mythical Man-Month*,[13] in part to dispel the notion that simply adding programmers could help speed up development time–in his experience, the converse was true, as more people only increased the complexity of the management and coordination task. Another managerial response to this crisis was the movement to define programming as a

[11] Nathan L. Ensmenger, "Letting the 'Computer Boys' Take Over: Technology and the Politics of Organizational Transformation," *International Review of Social History* 48, no. Supplement S11 (2003): 162.
[12] Ensmenger, "Letting the 'Computer Boys' Take Over," 163; Martin Campbell-Kelly and Aspray William, *Computer : a history of the information machine*, vol. 1, The Sloan technology series; (New York: Basic Books, 1996), 179-180.
[13] Frederick P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Anniversary ed. (Reading, Mass: Addison-Wesley Pub. Co, 1995).

"software engineering" discipline, to make it less of a craft more routinized and scientific.

The two historians who have written the most about the "software crisis" and the social and technological responses to it are Nathan Ensmenger, and the late Michael Mahoney. Mahoney has approached the topic from two angles, from the perspective of the quest for complete automation of programming, and from the perspective of software engineering. Mahoney lays out the history of software as a quest towards the Grail of the Automatic, Self-Programming computer,[14] an idea that was theoretically possible from the beginning in Turing's formulation of a universal finite state machine, which could replicate the functions of any machine through self-programming, limited only by memory. The development of symbolic assemblers, high-level programming languages and compilers (originally called "automatic programming") and operating systems increasingly automated many of the repetitive and tedious lower level tasks of dealing with the machine, and introducing increasing layers of abstraction, allowing programmers to focus on questions of how best to design software, and make software design, "best suited to various domains of application."[15] The "software crisis," generated two responses: one, to achieve full automation of programming, and two, to discipline the programmer, the former based on the Fordist assembly line, the later, the Taylorist "software factory." Mahoney describes the development of assemblers, high-level languages, and operating systems mostly as moves towards automatic programming, but they were also equally technologies that disciplined programmers by introducing ideas of new "best practices." Because software was, and still is, a "labor-intensive form of craft

[14] Mahoney, "From 0 to 1."
[15] Ibid., 96.

production",[16] the solution seemed straightforward: industrialize. Mahoney relates a famous quote by M.D. McIlroy:

> We undoubtedly produce software by backward techniques. We undoubtedly get the short end of the stick in confrontations with hardware people because they are the industrialists and we are the crofters. Software production today appears in the scale of industrialization somewhere below the more backward construction industries. I […] would like to investigate the prospects for mass production techniques in software.[17]

Software engineering would be likened to mechanical engineering, and the solution would be interchangeable parts and mass production. But industrialization meant social change as well as technological change, in that it disciplined and standardized the work of programmers. GE's R.W. Bemer's notion of the "software factory" provided a disciplined and regimented environment that would reduce costs:

> It appears that we have few specific environments (factory facilities) for the economical production of programs. I contend that the production costs are affected far more adversely by the absence of such an environment than by the absence of any tools in the environment… A factory supplies power, work space, shipping and receiving, labor distribution, and financial controls, etc. Thus a software factory should be a programming environment residing upon and controlled by a computer. Program construction, checkout and usage should be done entirely within this environment. Ideally it should be impossible to produce programs exterior to this environment…Economical products of high quality […] are not possible (in most instances) when one instructs the programmer in good practice and merely hopes that he will make his invisible product according to those rules and standards. This just does not happen under human supervision. A factory, however, has more than human supervision. It has measures and controls for productivity and quality.[18]

Mahoney sees this need for managerial control over programmers as a form of Taylorism, in its emphasis on the "supervision and support of the programmer […] with management seeking to impose the 'one best way' over a worker still in control of the

[16] Ibid., 94.
[17] Michael S. Mahoney, "Finding a history for software engineering," *Annals of the History of Computing, IEEE* 26, no. 1 (2004): 11.
[18] Ibid., 12-13.

shop floor."[19] Structured programming, a methodology of separating out different stages in the design process for different abstraction layers for software, was one such disciplining technology. Taylorism, as a management technology, was not the only model of industrialization, however. Many were attracted to the more explicitly technological approach of the assembly line.

> The evocation of the assembly line linked the software factory to a model of industrial production different from Taylor's—how different is a complex historical and technical question—namely Ford's system of mass production through automation. Ford did not have to concern himself about how to constrain workers to do things in "the one best way." His production machines embodied that way of doing things; the worker had little to do with it. The same was true of the assembly line. [20]

Software technologies that implemented the idea of interchangeable parts included "'mass-produced software components,' modular programming, object-oriented programming, and reusable software."[21]

Such technologies have not removed the programmer from the equation, however, nor have they even succeeded in eliminating its craft aspects. "Thirty years after the first NATO Conference on Software Engineering, advocates of an industrial approach to software development still complain that the 'vast majority of computer code is still handcrafted from raw programming languages by artisans using techniques they neither measure nor are able to repeat consistently.'"[22] However, James Tomayko argues that "software engineering" can be legitimately seen as engineering not because of its basis on

[19] Ibid., 13.
[20] Ibid., 15.
[21] Mahoney, "From 0 to 1," 99.
[22] Nathan L. Ensmenger and William Aspray, "Software as Labor Process," in *History of Computing: Software Issues*, ed. Ulf Hashagen, Reinhard Keil-Slawik, and Arthur L Norberg (Berlin: Springer, 2002), 139, quoting W. Wayt Gibbs, "Software's Chronic Crisis," *Scientific American*, September 1994.

underlying science but because its practitioners do engineering activities.[23] If so, software engineering shares with other kinds of engineering the qualities that it is both an art and a science. The fact that there are now handbooks and other sources of quantified engineering knowledge that can be taught to new practitioners is a signal that software engineering is maturing as an engineering discipline. However, in all engineering disciplines, handbooks only capture knowledge acquired through successful art, and serve to enable average engineers to be useful by allowing them to apply engineering principles that others have created. The reason for the persistence of art in engineering is the result of the "creativity gap"[24] that separates ideas from reality. Successful design of artifacts that can bridge the creativity gap require an intuitive "feel" for making the correct tradeoffs. [25] Indeed, Mahoney himself argues that software programming is more fruitfully seen not as "engineering" but as "architecture" because of that field's emphasis on design. To some extent, people in the software industry corroborate this, in that "software architecture" is a term used to refer to systems design.[26] Fred Brooks, who claimed that programming was like writing poetry, asserted in his 1987 article, "No Silver Bullet," that software could never be fully industrialized because its biggest problems lay in design, not implementation, and because of its socially embedded nature:

[23] James E. Tomayko, "Software as Engineering," in *History of Computing: Software Issues*, ed. Ulf Hashagen, Reinhard Keil-Slawik, and Arthur L Norberg (Berlin: Springer, 2002), 68.
[24] Tomayko borrows the term "creativity gap" from Louis Bucciarelli's ethnography of engineers: Louis Bucciarelli, *Designing engineers* (Cambridge Mass.: MIT Press, 1994), 76.
[25] Tomayko, "History of Computing: Software Issues," 67-68.
[26] Cox, "Planning the Software Industrial Revolution," 28.

to be successful, it must be compatible with the existing socio-technical context, yet also be able to grow and adapt to ever changing environments and user demands.[27]

Ensmenger agrees with Mahoney that the history of software is the history of its communities of practice, [28] or in other words, its programmers, although he acknowledges that the ambiguity of that term makes it more useful as a rhetorical device rather than an analytical category.[29] Like Mahoney, he sees responses to the software crisis as essentially a problem of labor and management, but he goes further. The articulation of the crisis itself is to be seen as a discursive construct of management. The software crisis was widely understood by managers as a labor shortage. However, the real problem was not "so much a lack of programmers *per se;* […but] a shortage of experienced, *capable* developers."[30] He finds that early in the crisis, it was recognized that in fact there was an oversupply of unproductive programmers, and that "exceptional" programmers were 10 times more productive than average ones, making the crisis about the shortage of "exceptional" programmers. However, the software community did not necessarily agree on what qualities made up the "exceptional" programmer.[31] Mathematical training in formal logic and numerical analysis, the original criterion for qualification as a programmer, was found to be of limited use in the business applications driving the need for more programmers.[32] The fluid boundaries of the programming workforce and the difficulty of agreeing on certification and training criteria became an obstacle to

[27] Frederick P. Brooks, "No Silver Bullet Essence and Accidents of Software Engineering," *Computer* 20, no. 4 (1987): 11-12.
[28] Mahoney, "What Makes the History of Software Hard."
[29] Ensmenger, "Letting the 'Computer Boys' Take Over," 154.
[30] Ensmenger and Aspray, "History of Computing: Software Issues," 141.
[31] Ibid., 141, 143.
[32] Ibid., 143.

professionalization.[33] Programmers had a harder time rising up to management than their colleagues in other engineering disciplines. However, the labor shortage worked in their favor, as the high demand meant for easy mobility for good programmers, driven to find jobs based more on job satisfaction than salary.

Much of the literature on the software crisis is written from the perspective of management. From that perspective, the situation did indeed seem like a crisis as the majority of computer costs by the end of the 1960s were for software production and most of that cost was labor. Disciplining programmers (often by technically unproficient managers) thus became a major priority. Object-oriented programming, for instance, was seen by one adherent as a way to impose the traditional workplace relationships of manufacturing. Taylorism was implicit in all the proposed solutions to the crisis.[34] Ensmengner and Aspray organize proposed "silver bullet" strategies to routinize, deskill, and control independent and recalcitrant programmers into three categories: *procedural structures, professional structures,* and *technological structures*.[35] The hyped methodology of object-oriented programming (OOP) in the 1980s was merely the latest in a series of methodologies attempting to reduce the "black art of programming" to a modern industrial manufacturing process. Brad Cox, an OOP advocate and primary creator of the Objective-C programming language, disputed Brooks' claim and asserted that OOP was the "silver bullet" to bring about the "industrial revolution" in software manufacture.[36] Nevertheless, Ensmenger argues that despite all these efforts, managers' efforts to completely routinize and de-skill programming tasks were unsuccessful; Philip

[33] Ibid., 147.
[34] Ibid., 151.
[35] Ibid., 153-157
[36] Ibid., 154.

Kraft's application of Harry Braverman's deskilling thesis to software ignores what Tomayko has corroborated as the "creative and intellectually demanding" aspects of programming. Management claims for silver bullets reflect more ideals than reality, as even Cox acknowledges that the crisis has not been solved.[37]

Mahoney and Ensmenger merely seem to differ on whether object-oriented programming was a Taylorist or Fordist technology of disciplining software workers. I lean towards Ensmenger's interpretation, as my experiences as a software worker on Apple's object-oriented Cocoa technologies suggest that OOP's proponents argue that it is useful primarily to discipline programmers into making better design choices, altering their practices. But was object-oriented programming, as both Mahoney and Ensmenger claim, created as a technology for managers to control their programmers? A closer look at some of the actors' accounts, both contemporary and retrospective, seem to indicate otherwise. For one, although Fred Brooks' *Mythical Man-Month* was originally published in 1975, and the first NATO Conference on Software Engineering took place in 1968 and 1969, the "silver bullet" debate between Brooks and Brad Cox took place between 1987 and 1990. Of course, that this occurred almost two decades after the "software crisis" was first perceived to have begun, illustrates the continuing problems that the industry has with the complexity of software and the craft nature of its production. Nevertheless, by 1990, object-oriented programming is not exactly new, although it is only around this time that it begins to gain widespread acceptance in the industry. What academic

[37] Cox writes in 1995: "The software crisis has been with us now for nearly a quarter-century [… Despite revolutionary advances in hardware], software remains intractable; immune to the organizational innovations that make us think of computer manufacturing as a mature industrial age enterprise.Brad J. Cox, "No Silver Bullet Revisited ," *American Programmer Journal* (November 1995), http://virtualschool.edu/cox/pub/NoSilverBulletRevisted/

computer scientists consider the first fully object-oriented high-level language, Smalltalk, was developed mostly by Alan Kay in various forms at Xerox PARC between 1970 and 1978.[38] C++, probably the most widespread and popular object-oriented language, was developed in various forms by Bjarne Stroustrup at Bell Labs between 1979 and 1986.[39] Cox, developed his own language, Objective-C, in the early 1980s as well.

Were these language designers responding to industrial managers when they invented these languages? Of the three, Cox seems to fit most closely to Ensmenger's thesis. Cox argued in the pages of IEEE Software that object-oriented programming would initiate the "software industrial revolution,"[40] a Kuhnian paradigm shift in the way software is produced, specifically echoing the sentiments of McIlroy. It was essentially the same argument as in his "There is a Silver Bullet"[41] response to Brooks, published in *Byte* magazine in the same year, 1990. Despite his historical references to Kuhn, the development of interchangeable parts in American gun manufacture,[42] and his persistent metaphor of industrial revolution, Cox did not believe that software production could be fully automated nor deskilled. In a subsection titled "Software Architecture," he explains:

> It is easy to see how interchangeable parts could help in manufacturing. But manufacturing involves replicating a standard product, while programming does not. Programming is not an assembly-line business but a build-to-order one, more akin to plumbing than gun manufacturing. But the principles of standardization and interchangeability pioneered for

[38] Alan C. Kay, "The early history of Smalltalk," in *The second ACM SIGPLAN conference on History of programming languages* (Cambridge, Massachusetts, United States: ACM, 1993), 69-95.
[39] Bjarne Stroustrup, "A history of C++," in *The second ACM SIGPLAN conference on History of programming languages* (Cambridge, Massachusetts, United States: ACM, 1993), 271-297.
[40] Cox, "Planning the Software Industrial Revolution."
[41] Cox, "There Is a Silver Bullet: A software industrial revolution based on reusable and interchangeable parts will alter the software universe."
[42] Cox, "Planning the Software Industrial Revolution," 25-26.

> standard products apply directly to build to-order industries like plumbing. They enabled the markets of today where all manner of specialized problems can be solved by binding standardized components into new and larger assemblies.
>
> Mature industries like plumbing are less complex than ours, not because software is intrinsically more complicated, but because they and not we have solved their complexity, nonconformity, and changeability problems by using a producer/consumer hierarchy to distribute these problems across time and organizational space. The plumbing supply market lets plumbers solve only the complexities of a single level of the producer /consumer hierarchy without having to think about lower levels, for example, by reinventing pipes, faucets, thermostats, and water pumps from first principles.[43]

Although Cox was explicitly responding to the perceived persistence of the software crisis, as a programmer he did not intend to propose a solution solely for the benefit of managers. In fact, Cox's own identity was ambiguous: he has both founded his own company, Stepstone, and has spent considerable time at George Mason University as a professor of computer science and mathematical biology. Rhetorically, he couched his prescriptions as benefiting "users," but more often "consumers":

> Furthermore, the intangibility imperative concentrates power in the hands of those with the abstract reasoning skills to comprehend an intangible product: the producers. The consumers are left powerless, unable to contribute the financial and legal resources that are needed to drive deep-seated cultural and technological changes.
>
> This drive to empower the consumer by making software as accessible and immediate as everyday tangible objects underlies the recent enthusiasm for direct manipulation (iconic) user interfaces, browsers, personal workstations, and other techniques for making software more tangible, less abstract, and more approachable by non programmers.[44]
>
> I use a separate term – software industrial revolution - to mean what "object- oriented" has always meant to me: transforming programming from a solitary cut-to-fit craft into an organizational enterprise like manufacturing. This means letting consumers at every level of an organization solve their own software problems just as home owners solve plumbing problems: by assembling their own solutions from a robust

[43] Ibid., 28-29.
[44] Ibid., 26.

> commercial market in off-the-shelf subcomponents, which are in turn supplied by multiple lower level echelons of producers.[45]

This supports Ensmenger's argument in the sense that we can see how, in Cox's market oriented rhetoric, "consumers" of software are likely to be corporations. Indeed, he almost reveals this when he says that it is the consumers who "pay our salaries." On the other hand, his reference to graphical user interfaces making computers more approachable to non-programmers is a reference to the new technologies of personal computing. Certainly, these work to the benefit of non-programming managers, but by 1990, a market exists for both home computers and packaged software to run on them. Cox's readership, in IEEE, is more likely to be the professional programming community than its managers, unless those managers are themselves former programmers. The evidence is ambiguous; it definitely supports Ensmenger's argument in that consumers are most likely to be imagined as "corporate customers," but on the other hand the audience is not managerial. In a fascinating turn, Cox's concern for "consumers" led him into a consideration of social construction:

> Lakoff and other authors in the collection that follows show why this approach fails as soon as objects leave the hands of their producer and encounter consumers with other views of the world. Categories (i.e. classes) cannot be invarients [sic] determined statically by the producer. They are determined (or better, must be determined once software engineering gets its poop in a group) by the consumer's worldview and culture.[46]

Histories of OO Languages, and the Static versus Dynamic controversy

Standard histories of software are often told as the development of increasing levels of abstraction and complexity, highlighting the shift from assembly level

[45] Ibid., 27.
[46] Brad J. Cox, "Social Construction of Reality," March 28, 2004, http://www.virtualschool.edu/mon/SocialConstruction/index.html.

programming to the adoption of higher level programming languages such as FORTRAN, COBOL, and C.[47] If oversimplified, the narrative becomes a teleological one where initial resistance by programmers preferring assembly gives way to adoption of the technologically "superior" solution of higher abstraction. A closer look reveals the persistence of schemes allowing programmers to circumvent such abstractions. As computer hardware understands only binary numbers (bits) and a few fundamental operations on them, the notion of software itself is based on hierarchical layers of abstraction that increase in complexity. For programmers, the higher the level of conceptual abstraction, the less control the programmer has over the hardware and what the computer actually has to do to execute the program, allowing programmers to construct more complex software at the cost of performance (less efficient use of the hardware in terms of speed of code execution by the CPU, or excessive memory usage). Many high level languages have been invented by academic computer scientists primarily concerned with mathematical theory, often with access to large mainframes or powerful workstations, resulting in languages with very high levels of conceptual abstraction. Working programmers must usually face the constraints of programming for the hardware limits faced by corporate customers and consumers, often using personal computers, which places priority on performance and efficiency. Indeed, the early high level, compiled languages faced resistance from working programmers who preferred hand-coding in assembly for precisely this reason. Thus, industrial or corporate designers of languages, faced with different concerns and constraints, have often arrived at designs

[47] Paul E. Ceruzzi, *A history of modern computing*, History of computing; (Cambridge, Mass. : MIT Press, 1998), 85-104; Campbell-Kelly and William, *Computer : a history of the information machine*, 165-172, 176-182.

that, from an academic perspective, "compromise" theoretical abstractions with practicality, allowing programmers some access to functionality closer to assembly language. Ironically, however, many of the technical responses to the software crisis, including the development of structured programming and object-oriented programming languages, are impositions of higher levels of abstractions that try to discipline programmers away from low level manipulation of the hardware, because it was precisely this low level manipulation that made programming a "black art" requiring high skill and creativity. Even with these new technologies, programmers could circumvent them if they wanted, and the programming community itself began to see a need to discipline programmers away from such harmful practices; structured and object-oriented methodologies could not merely be enforced by the language but had to be taught as a way of thinking.[48] The following section illustrates the trade-off and continuing tension between higher and lower levels of abstraction through a consideration of the history of two object-oriented programming languages, Smalltalk (and one of its conceptual heirs, Objective-C), and C++.

In order to understand the design choices involved, a brief technical overview of object-oriented programming is in necessary. (The following is based on explanations given in *Pitfalls of Object-Oriented Development* and an Apple updated version of an old

[48] A canonical example of this was the controversy over the *go to* statement in many procedural languages that circumvented structured programming, making program flow more random and thus harder to read and understand, more like an assembly program than one written in a higher level language. Ceruzzi, *A history of modern computing*, 103-104

NeXT Objective-C language reference.)[49] Procedural programming languages such as FORTRAN, COBOL, Pascal and C, which were the most widely used type of high-level languages, especially in industry prior to object-oriented languages, separate data and operations on data. Operations in such languages are usually known as procedures or functions. Object-oriented languages group data (state) together with the operations that act on it (behavior) into a single modular unit called an object. An object's data is known as its instance variables (or data members in C++) and the functions that operate on it are known as methods (or member functions in C++). In such languages such as Smalltalk and Objective-C, objects are seen as actors, which send messages to other objects, telling them to perform the actions (invoke the methods) specified by the message. The sending object has no idea how the receiver implements the behavior asked for. The black-boxing of an object's internal implementation from its interface to the outside world is a fundamental feature of object-oriented programming known as "encapsulation" or "information hiding." This creates a barrier between an object's data and external entities that might try to modify it without permission, enforcing its modularity and thus reusability. Both instance variables and method implementations are usually protected in this manner. Interfaces and implementations are defined in an object's "class." Individual objects are all instances of the same kind of object, its "class." A class is a template for a kind of object. Many individual objects of a class can be created, or "instantiated" and

[49] Apple Computer, Inc., *Inside Cocoa: Object-oriented programming and the Objective-C language* (Cupertino CA: Apple Computer, 2000); Webster, *Pitfalls of object-oriented development*.

different data values can be assigned to each object's instance variables. A class is a data type like Integer or Array, but custom defined by the programmer.[50]

All fully object-oriented languages feature "polymorphism." Objects of different classes may have methods with the same names, such as "draw." This allows a message sender to tell an arbitrary object to simply draw without knowing in advance what class of object it is receiving the message. Each class capable of responding to the message "draw" has its own method implementation of the action to be performed. This implementation can be radically different; an example is that a circle class and a square class would draw themselves differently. The abstraction of polymorphism helps enforce the modularity and flexibility of programs by not requiring senders to know in advance all the possible actions all possible receivers should implement.

Finally, all object-oriented languages feature "inheritance." New class definitions can be based on previously defined classes, called superclasses, parent classes, or base classes in C++. The inheriting class is called a subclass, child class, or derived class in C++, and inherits all of its parent's instance variables and methods. It can define additional instance variables and methods that the superclass did not have, and it can override its parent's implementation of specific methods by redefining them. For example, a square class can inherit from a rectangle class and redefine its draw method. In this sense whole trees or hierarchies of super and subclasses are created, with the classes at the root of the tree the most abstract. Some object-oriented languages such as C++ have multiple inheritance, allowing a class to derive from multiple superclasses. Objective-C and Java have only single inheritance in the strict sense, but allow

[50] Apple Computer, Inc., *Inside Cocoa: Object-oriented programming and the Objective-C language*, 15-28.

inheritance of method implementations (but not instance variables) from multiple superclasses through features such as “protocols” and “categories” in Obj-C and “extensions” in Java.

All languages considered fully object-oriented feature encapsulation, polymorphism, and inheritance. However, there is a general differentiation between static languages such as C++, and dynamic languages, which include Smalltalk, Objective-C, Ruby and Python. Dynamism essentially means not locking in certain decisions in the program at compile and link time (when the programmer tells the computer to translate the source code to an actual running program),[51] but dynamically at runtime in order to respond to user input or changes in the environment. There are multiple forms of dynamism, but for our purposes it is only important to understand dynamic typing and dynamic binding, which are closely related but often confused with each other. Static typing requires the programmer to set the type of an object in advance, which has the advantage of allowing the compiler to catch type errors for the programmer and do some performance optimizations. Dynamic typing means that the object’s type is not decided until the point during program execution when it becomes necessary to know, which usually requires a generic type that can stand for any class. What might be considered an

[51] Source code in a human readable high level programming language is first “compiled” into “object code,” which is binary machine code the computer can execute. The object code is usually compiled into multiple “objects,” each one associated with a single or small group of source code files. Because most programs are composed of multiple groupings of source files, these separate “object” files must be linked together, and to any standard libraries of prepackaged functionality provided by the operating system or programming language environment, in order to run correctly. Lastly, before a program can be run, the linked object files must be loaded into memory. A third form of dynamism involves this “loading stage,” allowing pieces of the program, extensions to it, or libraries that it depends on, to not be loaded all at once but as needed at runtime, saving memory and increasing flexibility and extensibility of the program, even possibly at the request of the user.

error for a statically typed language is fully legal in a dynamic language and can be exploited to do things that would be difficult to accomplish in a statically typed language. Static binding requires the compiler to bind the specific method that will get invoked after a message is sent to an object when the program is compiled, which again, has both reliability and performance advantages, while dynamic binding delays this decision until runtime, requiring the runtime system to dynamically dispatch the message to an object which can properly receive it, requiring additional overhead.[52] The benefit of this overhead is additional flexibility. Apple's Objective-C reference polemically argues that full polymorphism requires the use of both dynamic typing and binding, which becomes crippled if only one or neither are used.[53] Dynamic binding is sometimes called late-binding, including by Alan Kay, although Apple's reference argues that late-binding refers only to a compromise measure to achieve polymorphism in statically typed languages such as C++. In C++, an object's type, though static, can be inexact: it can be of its own class or any superclass it inherits from. However, because of this, the compiler cannot know which version of a method to bind because it does not know the class of the message's receiver. If a method in C++ is declared "virtual," this decision is postponed

[52] Webster, *Pitfalls of object-oriented development*, 17

[53] "Dynamic binding is possible even in the absence of dynamic typing, but it's not very interesting. There's little benefit in waiting until run time to match a method to a message when the class of the receiver is fixed and known to the compiler. The compiler could just as well find the method itself; the run-time result won't be any different.
"However, if the class of the receiver is dynamically typed, there's no way for the compiler to determine which method to invoke. The method can be found only after the class of the receiver is resolved at run time. Dynamic typing thus entails dynamic binding.
"Dynamic typing also makes dynamic binding interesting, for it opens the possibility that a message might have very different results depending on the class of the receiver. Run-time factors can influence the choice of receiver and the outcome of the message." Apple Computer, Inc., *Inside Cocoa: Object-oriented programming and the Objective-C language*, 38

until the program is linked, though not all the way till runtime, which means that user actions cannot influence this decision.

Most of these metaphors and terminology are from Alan Kay's Smalltalk, considered the first fully object-oriented language. A comparative examination of the historical development of these Smalltalk and C++ will reveal very different priorities and assumptions that have consequences in their designs. Brad Cox's hybrid Objective-C language is basically a reimplementation of Smalltalk concepts on top of the procedural language C, and thus, being a direct intellectual descendant, is more similar to Smalltalk than it is to C++, despite both hybrid languages sharing the same C base. Cox's overarching concern for shifting the balance of power towards the "user" and the "consumer" is also similar to Alan Kay's goals in designing Smalltalk. Kay does not fit Ensmenger's profile as a designer who worked for corporate or management interests. Before joining Xerox PARC, Kay's had come straight out of academia, working on ARPA projects at Utah under Ivan Sutherland. The team at Xerox PARC was composed largely of either former members of Doug Engelbart's human augmentation group at Stanford Research Institute (SRI) or Utah graduates who specialized in graphics. PARC, although being a corporate R&D lab, was much closer to the academic environments that Kay had been accustomed to, and indeed, Kay describes the constant conflicts they had with Xerox's corporate headquarters.[54] His design of Smalltalk both incorporates and reacts against features of LISP, the functional language used at Stanford's AI lab (SAIL).[55] Moreover, Kay's overarching concern is with his vision of "personal

[54] Kay, "The early history of Smalltalk," 13.
[55] Functional languages are another kind of high-level programming language different from both procedural and object-oriented languages.

computing" based on Engelbart's notion of "human augmentation."[56] Kay's ultimate goal is to design a computer for children, a "KiddiKomp," and the name "Smalltalk" reflects that.[57] The influence of Seymour Papert's LOGO, a language designed for learning in children, is also mentioned several times.[58] Kay's intended users for Smalltalk (despite his failure at actually getting them used by children), are decidedly not corporate managers. Nevertheless, in creating not just a language but whole new approach in programming, Kay created a system that was much closer conceptually to modeling the kinds of problems ordinary users might want their applications to solve than the lower level procedural languages. Such a system might make the system easier to program for anybody, be that person a corporate manager or a child.

While Kay had the luxury of remaining in an almost academic research environment, Bjarne Stroustrup, developing C++ at Bell Labs, was concerned primarily with the needs of the working programmer. In describing his design goals for C++, he revealed that it did not involve innovation, but focused on efficiency and flexibility.[59] As Stroustrup was concerned primarily with the languages' acceptance in the industry, and its utility for low-level systems programming which required fast-executing code, he did not consider his primary competition to be other object-oriented languages but plain C, a language whose high industry acceptance was seen to be in part because of its access to lower levels of the machine, allowing for programs that could approach the speed of handwritten assembly code. For C++ to succeed, it not only needed to maintain a significant amount of compatibility with C, but it had to generate code that ran as fast as

[56] Ibid., 7.
[57] Ibid., 14.
[58] Ibid., 10.
[59] Stroustrup, "A history of C++," 1.

C, just as C’s success was in part due to its success at generating code competitive with assembly languages. Any language features that might compromise this goal were omitted from C++. Stroustrup’s overarching concern with practicality and widespread adoption in the context of C’s market dominance explains C++’s popularity today among the vast majority of working programmers. It also explains, however, some of the criticisms that users of object-oriented languages such as Smalltalk and Objective-C have of C++. Stroustrup reveals that initially C++ began as merely an attempt to add “abstract data classes,” to C, which would improve code structure. Stroustrup saw the class hierarchy, or inheritance, as the only necessary feature for a language to be considered “object-oriented.”[60] Features necessary to support polymorphism, especially late binding, were added incrementally over time: virtual functions in 1984’s first release of C++, abstract classes in 1986’s 2.0, parameterized types in the form of templates not until 1988.[61] Stroustrup’s concern with efficiency led him to emphasize strong, static (compile-time) typing. While he acknowledged some of the flexibility advantages of dynamic typing at runtime (acknowledging that it has “deep implications for the way one designs systems”),[62] the need to keep execution fast, as well as to help the programmer avoid type related errors, led to a consistent commitment to static typing. This meant that rather than being designed in from the beginning, features conducive to dynamic-style programming were added incrementally, and added to the complexity of the language. Contrast this with Kay, who considers dynamic binding to be the fundamental feature of

[60] Ibid., 30.
[61] Templates are necessary in C++ because, as a statically typed language, it cannot assign variables to objects of a generic class, which is useful for algorithms that operate on collections of multiple objects.
[62] Ibid.

object-oriented programming: "OOP is a late binding strategy for many things…"[63] "OOP can be viewed as a comprehensive technique for late-binding as many things as possible…"[64]

Why is this rather technical discussion on the debate between "static" versus "dynamic" typing and binding interesting? Both C++ and Objective-C are hybrid languages, object-oriented extensions built on top of the low level, procedural language C. This allowed programmers the flexibility to both work at higher levels of conceptual abstraction while also being able to "go down to C" when necessary for performance reasons. Both Stroustrup and Cox claimed that they got the best of both worlds with their hybrids,[65] but they differed on a fundamental design choice: C++ prefers static typing and only "late," link-time binding,[66] while Objective-C prefers dynamic typing and binding at runtime. Cox's notion of the right trade-off to make between static and dynamic binding

[63] Kay, "The early history of Smalltalk," 25.
[64] Ibid., 39.
[65] "Having both static type checking and dynamic type identification (for example, in the form of virtual function calls) implies some difficult tradeoffs compared to language with only static or only dynamic type checking. The static type model and the dynamic type model cannot be identical and thus there will be some complexity and inelegance that can be avoided by supporting only one type model. However, I wouldn't want to write programs with only one model." Stroustrup, "A history of C++," 48
"Strongly coupled languages like Ada and C++ are deficient insofar as they do not also support loosely coupled modularity /binding technologies like screwing, bolting, welding, soldering, pinning, riveting, and mortising. And loosely coupled environments like Smalltalk are deficient insofar as they fail to support tightly coupled modularity/binding technologies like forging and casting. Hybrid environments like Objective-C and CLOS, and analogous ones that could be based on Cobol, Ada, Pascal, or C++, support multiple modularity binding technologies as tools to be picked up or laid aside according to the job at hand." Cox, "Planning the Software Industrial Revolution," 28
[66] "Firstly, I considered and still consider static type checking essential both as a support for good design and secondarily for delivering acceptable run-time efficiency." Stroustrup, "A history of C++," 48
Late binding in C++ is also not on by default, but must be explicitly declared for a member function with the "virtual" keyword. In Objective-C, all methods are implicitly dynamically bound, there is no static option.

was based on what level the software corresponded to on a hardware manufacturing metaphor. The hardware community's industrialized advantages needed to be emulated by making software components interchangeable parts. At a certain level, dynamic binding was critical to this goal. "The hardware community's monumental achievements are largely due to the division of labor made possible by the loosely coupled modularity/binding technologies shown in this figure. Card-level pluggability lets users plug off-the-shelf cards to build custom hardware solutions without having to understand soldering irons and silicon chips. (Pluggability is the ability to bind a component into a new environment dynamically, when the component is used, rather than statically, when it is produced.)"[67] This was critical to Cox's project of making software serve the end-user by making it flexible, rather than Stroustrup's concern with programmer acceptance through code speed. For Cox, if a programmer needed to trade off user flexibility for speed, Objective-C made this possible by simply including all of C as part of the language. Cox considered this to be the strength of his hybrid approach, but because the Smalltalk-like syntax of the object-oriented portions of the language and the procedural C portions were completely orthogonal to each other, switching between levels of abstraction in the hierarchy became very explicit. C++'s approach was much more incremental–object-oriented concepts accrued in the language slowly over time, and the syntax deliberately was made to not be a radical break from procedural C. While this made the new OOP concepts easier to digest for procedural C programmers, Objective-C advocates could argue that this meant that most C++ programmers did not truly understand what the OOP approach was really about, especially if they agreed with Alan

[67] Cox, "Planning the Software Industrial Revolution," 29.

Kay that late-binding was critical to it.[68] Cox believed that the OOP community's support of strong typing (even in dynamically bound languages such as Java) favored software producers and not consumers: "I'd noticed breakdowns in the way inheritance and strong type-checking are viewed by the object-oriented programming language community. This community's established paradigm is that an object's 'class,' its 'category' acc'd [sic] Lakoff, should be determined by the object's producer in pursuit of implementation (how to build it) [sic] concerns, without regard to the objects use, its users, or the consumer's notions of what the object is 'for'."[69]

The controversy between dynamic versus static language advocates has a specific resonance with the Macintosh programming community. Apple's classic Macintosh platform, from 1984 to 2000, was based on a procedural applications programmer interface (API) known as the Macintosh Toolbox, originally allowing third party developers access through Pascal but later C and C++. In 1997, Apple acquired NeXT,

[68] To an extent, one can see this in the way C++ and Objective-C programmers refer to method invocation. C++ methods are called "member functions," retaining procedural C's notion of the function, with the only difference being that they are now "members" of a class. In procedural languages, functions are "called." Smalltalk and Objective-C programmers insist that methods are never "called;" rather they get executed when a message gets passed to a receiving object. Due to dynamic binding, which method gets executed depends on what object received the message, which is determined as the program is running, at "run-time." Programmers new to Objective-C, often coming from Java, a more C++-like language syntactically, often use the phrase "call a method," but according to John Randolph, former Cocoa Evangelist for Apple, this is strictly incorrect. What is interesting for our purposes is not what utterance is "correct" or "incorrect," but rather that to some extent the Objective-C advocates claim that the procedural and object-oriented "paradigms" are incommensurable with each other. Programmers who still refer to "method calls" have not truly made the mental Gestalt switch to the new paradigm. C++ encourages this by encouraging programmers to continue to think in old procedural terms even as it tries to teach object-oriented concepts. As a result, Objective-C advocates claim that many C++ programmers do not truly understand object-oriented programming.

[69] Cox, "Social Construction of Reality."

whose Unix-based operating system, NeXTStep, became the basis of Mac OS X, the replacement for the classic Mac OS. NeXTStep's developer API was object-oriented and based on Objective-C, which NeXT had purchased from Brad Cox's Stepstone in the mid-1990s.[70] A small but passionate community of NeXT programmers, convinced of the technological superiority of dynamic object-oriented programming, grew up around the NeXTStep technologies through its various incarnations. After the Apple acquisition, the NeXTStep "framework" (an object-oriented API and set of shared libraries) was renamed "Cocoa."[71] However, in order to facilitate the transition to its new operating system for its third party developers, particularly its large corporate developers Adobe and Microsoft, Apple decided to create a version of its venerable procedural API, the Mac Toolbox, that would be "native" to Mac OS X, and renamed it, "Carbon." Developers for Mac OS X were given a choice between programming for Cocoa using Objective-C or programming for Carbon using C or C++, and this largely mapped onto a historical social division among developers. Carbon developers were usually longtime Macintosh developers with legacy code-bases rooted in the classic Mac OS, often large corporate developers such as Adobe. Cocoa developers were often old NeXT developers who had been with that platform through the 1990s, or possibly were developers new to the Mac and had little or no legacy code. Many of these Cocoa developers were "independent" developers or "indies," individuals or small entrepreneurial ventures formed by the developers themselves. In the first half of this decade, Apple's official line to its developers was that Carbon and Cocoa were peers with neither one given preference over

[70] Brad J. Cox, "Next Acquires Objective-C," http://www.virtualschool.edu/objectivec/nextAcquresOC.html
[71] The name was probably chosen to evoke comparisons to Java, Sun's much hyped new object-oriented language and environment at the time.

the other, and that Carbon would be supported indefinitely. In 2006, however, Apple announced that it was not going forward with a 64-bit version of Carbon in the next version of Mac OS X, which signaled to developers that Carbon's future was numbered. The reaction from the Carbon community was extremely hostile, inciting a highly polemical debate over the technical merits of Cocoa and Carbon between its partisans.[72] Apple's refusal to reconsider this decision over the last three years has led to some grudging closure within the Mac community, but resistance to Objective-C continues in other forms, with some developers claiming it is "ugly," and a "failed experiment,"[73] and others asserting it has a "steep learning curve."[74]

Conclusion and Future directions

This paper represents a first exploration into ideas for a dissertation that contains a mishmash of various ideas and projects, which are not as well-connected as they ought to be. In fact, this could very well be two separate papers, and in the future might possibly be separated. Presenting them here together, however, allows us to consider the connections between these topics, which, although loose, are there. Mahoney and Ensmenger present the literature on the software crisis, and I agree with them that the crisis was in part a discursive construction by middle managers to regain a measure of

[72] Daniel Jalkut, "Red Sweater Blog - The Cocoa-Carbon Advantage," blog, *Red Sweater Blog*, September 7, 2006, http://www.red-sweater.com/blog/181/the-cocoa-carbon-advantage; John Gruber, "Daring Fireball: Some Assembly Required," blog, *Daring Fireball*, October 5, 2006, http://daringfireball.net/2006/10/some_assembly_required; Scott Stevenson, "Theocacao: Some Explanation Required, Cocoa and Carbon," blog, *Theocacao*, October 6, 2006, http://theocacao.com/document.page/314
[73] Mike Fulton, "Objective C - A Failed Experiment," blog, *Mike Fulton Says…*, April 29, 2009, http://www.mikefulton.net/2009/04/29/objective-c-a-failed-experiment/
[74] Charles Babcock, "Startup's iPhone SDK Steers Clear Of Apple's Objective-C," *InformationWeek*, June 24, 2009, http://www.informationweek.com/news/personal_tech/iphone/showArticle.jhtml?articleID=218100982

control over a workforce that was increasing in bargaining power due to its monopoly over expertise. However, although they discuss object-oriented methods and technologies in this light, evidence from the second section points to a more complicated, and not completely clear, picture. The history of Smalltalk and C++ indicates that, from the perspective of the men who created the technology itself, their concerns were not exactly those of management. Furthermore, one would expect managers to favor such technologies that were more abstract and aid in the management of complexity, but the dominance of C++ over languages with full dynamic typing and binding shows that practical concerns of performance and efficiency, as well as not requiring newcomers to fully "paradigm-shift" into object-oriented methodology, proved more salient to industry-wide acceptance.

Furthermore, ethnographic research at an iPhone startup company that I undertook in 2008 showed that certain communities of programmers can be more militant about object-oriented principles and technologies than their managers, who care more about business priorities such as how much industry acceptance a particular language has and its proven track record, which might trump the marginal technical or managerial benefits of using a superior but more obscure language. In my case study, Cocoa programmers at the company pushed a stricter, more dynamic object-oriented language (Objective-C) for running their servers, over the objections of management, who preferred to use the industry standard, PHP, despite arguments that Objective-C would have lower costs and faster time to market. Business managers may be concerned with how a technology fits in with the larger socio-technical systems their particular product must interface with rather than specific technological benefits, especially if the alternative is seen as “good enough.” But what still seems paradoxical is why Cocoa programmers advocate Objective-C in the first place if such a technology was originally created to discipline and their work? There are some possible explanations. First, although object-oriented programming disciplines a programmer, such discipline is seen as beneficial to the programmer to avoid bad habits, ultimately freeing the programmer from tedious debugging and allowing for more creative, artistic work to take place. Second, our examination of Alan Kay’s motivations in creating Smalltalk suggests an ideological component to advocacy of object-orientation. Kay’s project was part of a larger “personal computing” movement associated with the 1960’s counterculture and with Doug Engelbart’s human augmentation project. Through Stewart Brand’s *Whole Earth Catalog*, a countercultural re-appropriation of cybernetics and small-scale digital calculators as tools not for military-industrial-bureaucratic control but for personal liberation and transcendence through information mastery lived on in the personal computing movement, replacing LSD with PCs as mind expanding tools.[75] Two key features of the Smalltalk system, the graphical user interface and object-oriented programming, both were intended to bring computing power to the masses. Seen in this light, advocacy of dynamic object-oriented programming, and Cocoa in particular, is an ideological project of conversion.

Significant future work needs to be done. Much of the literature on engineering-management relations complicates the picture set out here. Noble (1985) presents engineers as agents for managers trying to use computer technology to deskill workers, while Oldenziel (1999) presents them as ambiguous figures, carving professional

[75] Fred Turner, *From counterculture to cyberculture : Stewart Brand, the Whole Earth Network, and the rise of digital utopianism* (Chicago: University of Chicago Press, 2006)

identities vacillating between the masculinity of the shopfloor and the genteel qualities of the academic scientist and or white collar manager, with the lower echelons often reduced to routine calculation and only the higher levels achieving the promise of joining the ranks of management. The history of object-orientation, both as a set of technological artifacts in the form of languages, environments, and tools, and as a set of design methodologies, practices, and knowledge, needs to be explored further. Further analysis of *Pitfalls of Object-Oriented Development* (Webster 1995)[76] needs to be done to examine the arguments for "best practices" separating "pure" object-oriented designs from those which compromise the model, and exactly what kind of efficacy is achieved through discipline. Another influential book in OO design is Gamma, et. al., 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*,[77] which Webster claims is a "must-have."[78] Gamma et. al., offers still higher levels of abstractions as a toolkit for designers, which became heavily incorporated into Apple's Cocoa frameworks. Whole chapters could be written on the development of Smalltalk, C++, Objective-C, and other languages with access to primary sources. More in-depth analysis of Brooks' *Mythical Man-Month*, and of Brad Cox's "Silver Bullet" arguments, as well as other literature on software engineering, would flesh out the paper. Lastly, significant additional ethnographic data has been gathered on other companies, allowing for more comparison of management-engineering relations. This work can be multiple papers in its own right.

[76] Webster, *Pitfalls of object-oriented development*
[77] Erich Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley professional computing series (Reading, Mass: Addison-Wesley, 1995).
[78] Webster, *Pitfalls of object-oriented development*, 245

I began this paper by quoting Michael Mahoney, and it is fitting that to him I return. Mahoney said, "the history of software is hard," precisely because ultimately it is the history of human beings and how they mentally model the world. I do not know how closely I have followed this in this paper, especially in the more technical sections, but if I have failed, it is because the history of software is indeed hard. Yet I hope that at least on an abstract level, the spirit of Mahoney has guided me in this paper, and will continue to guide me in future endeavors.

References

Apple Computer, Inc. *Inside Cocoa: Object-oriented programming and the Objective-C language*. Cupertino CA: Apple Computer, 2000.

Babcock, Charles. "Startup's iPhone SDK Steers Clear Of Apple's Objective-C." *InformationWeek*, June 24, 2009. http://www.informationweek.com/news/personal_tech/iphone/showArticle.jhtml?articleID=218100982.

Brooks, Frederick P. "No Silver Bullet Essence and Accidents of Software Engineering." *Computer* 20, no. 4 (1987): 10-19.

---. *The Mythical Man-Month: Essays on Software Engineering*. Anniversary ed. Reading, Mass: Addison-Wesley Pub. Co, 1995.

Bucciarelli, Louis. *Designing engineers*. Cambridge Mass.: MIT Press, 1994.

Campbell-Kelly, Martin, and William Aspray. *Computer : a history of the information machine*. Vol. 1. The Sloan technology series;. New York: Basic Books, 1996.

Ceruzzi, Paul E. *A history of modern computing*. History of computing;. Cambridge, Mass. : MIT Press, 1998.

Cox, Brad J. "Next Acquires Objective-C." http://www.virtualschool.edu/objectivec/nextAcquresOC.html.

---. "No Silver Bullet Revisited ." *American Programmer Journal* (November 1995). http://virtualschool.edu/cox/pub/NoSilverBulletRevisted/.

Cox, Brad J. "Planning the Software Industrial Revolution." *IEEE Software* 7, no. 6 (November): 25.

---. "Social Construction of Reality," March 28, 2004. http://www.virtualschool.edu/mon/SocialConstruction/index.html.

---. "There Is a Silver Bullet: A software industrial revolution based on reusable and interchangeable parts will alter the software universe." *BYTE*, October 1, 1990.

Deitel, Harvey M., and Paul J. Deitel. *C++ : how to program*. Englewood Cliffs N.J.: Prentice Hall, 1994.

Ensmenger, Nathan L. "Letting the 'Computer Boys' Take Over: Technology and the Politics of Organizational Transformation." *International Review of Social History* 48, no. Supplement S11 (2003): 153-180.

---. *The "Computer Boys" Take Over: Computers, Programmers, and the Politics of Technical Expertise*. Cambridge, MA: MIT Press.

Ensmenger, Nathan L., and William Aspray. "Software as Labor Process." In *History of Computing: Software Issues*, edited by Ulf Hashagen, Reinhard Keil-Slawik, and Arthur L Norberg, 139-165. Berlin: Springer, 2002.

Fulton, Mike. "Objective C - A Failed Experiment." Blog. *Mike Fulton Says…*, April 29, 2009. http://www.mikefulton.net/2009/04/29/objective-c-a-failed-experiment/.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Reading, Mass: Addison-Wesley, 1995.

Gruber, John. "Daring Fireball: Some Assembly Required." Blog. *Daring Fireball*, October 5, 2006. http://daringfireball.net/2006/10/some_assembly_required.

Jalkut, Daniel. “Red Sweater Blog - The Cocoa-Carbon Advantage.” Blog. *Red Sweater Blog*, September 7, 2006. http://www.red-sweater.com/blog/181/the-cocoa-carbon-advantage.

Kay, Alan C. “The early history of Smalltalk.” In *The second ACM SIGPLAN conference on History of programming languages*, 69-95. Cambridge, Massachusetts, United States: ACM, 1993.

Mahoney, Michael S. “Computer Science: The Search for a Mathematical Theory.” In *Science in the Twentieth Century*, edited by John Krige and Dominique Pestre, 941. Australia: Harwood Academic Publishers, 1997.

---. “Finding a history for software engineering.” *Annals of the History of Computing, IEEE* 26, no. 1 (2004): 8-19.

---. “Software: The Self-Programming Machine.” In *From 0 to 1: An Authoritative History of Modern Computing*, edited by Atsushi Akera and Frederik Nebeker, 91-100. New York, N.Y: Oxford University Press, 2002.

---. “The History of Computing in the History of Technology.” *Annals of the History of Computing, IEEE* 10, no. 2 (1988): 113-125.

---. “What Makes the History of Software Hard.” *Annals of the History of Computing, IEEE* 30, no. 3 (2008): 8-18.

Noble, David F. “Social choice in machine design: the case of automatically controlled machine tools.” In *The Social shaping of technology : how the refrigerator got its hum*, edited by Donald MacKenzie and Judy Wajcman, 109-124. Milton Keynes ; Philadelphia: Open University Press, 1985.

Oldenziel, Ruth. *Making technology masculine : men, women and modern machines in America, 1870-1945*. Amsterdam: Amsterdam University Press, 1999.

Stevenson, Scott. “Theocacao: Some Explanation Required, Cocoa and Carbon.” Blog. *Theocacao*, October 6, 2006. http://theocacao.com/document.page/314.

Stroustrup, Bjarne. “A history of C++.” In *The second ACM SIGPLAN conference on History of programming languages*, 271-297. Cambridge, Massachusetts, United States: ACM, 1993.

Tomayko, James E. “Software as Engineering.” In *History of Computing: Software Issues*, edited by Ulf Hashagen, Reinhard Keil-Slawik, and Arthur L Norberg, 139-165. Berlin: Springer, 2002.

Turner, Fred. *From counterculture to cyberculture : Stewart Brand, the Whole Earth Network, and the rise of digital utopianism*. Chicago: University of Chicago Press, 2006.

Webster, Bruce. *Pitfalls of object-oriented development*. New York: M&T Books, 1995.