An Open Letter to the Software Nobility

By Geoff Cohen & Alan Radding

January, 2002

An Open Letter to the Software Nobility

by Geoff Cohen & Alan Radding

"Now, although I am too small a man to make propositions which might effect a reform in this dreadful state of things, nevertheless I may as well sing my fool's song to the end, and say, so far as I am able, what could and should be done . . ."

Martin Luther, *An Open Letter to The Christian Nobility of the German Nation Cencerning the Reform of the Christian Estate, 1520*

Martin Luther struggled against a single, monolithic church that maintained absolute standards and laws. And although his goal was to reform this single church, his protest instead sparked the creation of hundreds of competing churches, each with incompatible doctrines, rules, and cultures. The subsequent conflict took tens of thousands of lives and devastated Europe for a generation. Today, however, most of us have learned to live with many churches and faiths, each serving its own community in its own ways.

The world of software—far younger than Catholic church in the 1500s, though no less dogmatic—is as fragmented as the post-Reformation churches, and as prone to religious wars over languages, methodologies, operating systems, protocols, tools, and the like. In the process, the software world has spawned a vast amount of criticism, complaint, frustration, anger, resentment, envy, greed; a veritable litany of sins. Fifty years after people started programming commercial computers, the applications are still late, costly, bloated, inflexible, hard to use, unreliable, flawed. And the slew of paradigms and approaches spanning the cathedral to the bazaar have failed quell the rumblings.

The Cap Gemini Ernst & Young Center for Business Innovation recently gathered twenty leading thinkers to focus on the future of software. Although the group did not leave with 95 theses to nail on the IT department door, it did identify three key changes that will redefine how software is created and used:

- the abandonment of top-down control as the central dogma of computation;
- shifts in the balance of responsibility between users and software creators; and
- the legitimization of alternate models of computation beyond the Turing machine.

The common thread that unites these changes is an increasing diversity in approaches, applications, users, and architectures. And although diversity often comes at a cost of increased maintenance and coordination cost, ultimately, there is no one way to create software any more than there is only one church.

In 1976, Bill Gates sent an "Open Letter to Hobbyists," calling on personal computer owners to stop stealing his software. No one will write software for free, he claimed, and thus heralded the dawn of the age of commercial PC software. Perhaps our open letter—this fool's song—may in some small way signal another age of software, one with broader participation, changed assumptions, and new metaphors. And although we can only roughly describe the changes that will bring in this new age, we believe that they are necessary, and that they are coming.

This Dreadful State of Things

If, as some assert, creating software is an engineering discipline, it should be orderly and predictably consistent, and at certain times and in certain situations it is. However, the strict application of the engineering metaphor can lead one astray. When software is designed, it is a model. It may model a business process, or the earth's atmosphere, or a document. But, as Gregor Kiczales of the University of British Columbia points out, when it is completed and used, in many cases that model, rather than mirroring reality, actually controls and defines it. Actions can't be taken if they weren't anticipated when the model was created ("I'm sorry, the system won't let me do that."). And while this software-imposed and engineered reality offers consistency and order, that comes at a cost of brittleness, inflexibility, and constraint.

The engineering metaphor faces increasing dissonance between model and reality. Until quite recently, the vast majority of programming work assumed a single computer, with a reliable memory, disk, and processor, with a predictable environment and implicitly authorized users. These assumptions were good for the lifetime of the software, and good enough to let the engineering approach at least muddle through.

The world, however, is rapidly turning into a very different place. A myriad of computers and environments communicate and collaborate using dozens, if not hundreds, of different protocols. Remote resources can no longer be predicted, not to mention trusted. And without warning, interfaces and behavior change, evolve, or even go away entirely. As independent consultant and researcher Clay Shirky emphasizes, the assumptions and techniques of the world of the single box simply no longer apply to software functioning at internet scale.

The difference is more than merely cosmetic or even just a matter of scale. Frederick Brooks, in *The Mythical Man-Month*, offered a factor-of-three rule of thumb to estimate the increase in complexity at each stage as one progressed from simple programming to programming systems, and then to programming products. The leap from the programming system products Brooks described—products still contained within a well-defined and controlled box—to programming solutions for the distributed, wildly interconnected and dynamic environment must represent a gain in complexity far beyond a factor of three.

Our first impulse is to deal with these problems the same way we dealt with old problems: by trying to contain and manage this anarchy through the use of such techniques as comprehensive global standards and well-defined protocols, in effect just building a much bigger single box. This approach has its merits. A certain amount of standardization is necessary and desirable. It allows trains to move smoothly from one set of tracks to another, and builds enough confidence so that people will be willing to invest time and money in building on top of that standard foundation. Without any standardization, interoperability would be an unreachable goal.

However, in the network age, software standards inevitably fall short. The natural anarchy of software undermines adoption of 'universal' standards. Variations in the way people implement a standard, or unreliable services, or malicious intent can break the system. Worse, even when successful, overreaching standards constrain innovation, limit freedom, and lock applications and users into an increasingly

false model of reality. In the end, we've networked Pandora's box, and we can't wish the new world back the way it was.

Yet anarchy in software is not such a bad thing. It is the very anarchy of software, with its alphabet soup of competing standards, that allows it to adapt to dynamic reality, to escape the constraints of engineered reality. And so although this anarchy makes software less reliable, buggier, and more unpredictable, we should welcome it. We ought to: we don't have a choice. And once we allow for the fact that software doesn't have to be perfectly repeatable or predictable, all sorts of things are possible.

Repent and Accept Complexity Into Your Heart

In the past two decades, the science of complex adaptive systems (CAS), also known as complexity science or chaos theory, has drawn on biology and physics to produce a new model of how order can emerge from distributed action. This science may offer a way forward for a new way to construct software. According to John Casti of the Santa Fe Institute, a typical CAS consists of a collection of autonomous agents, ranging from a few hundred to a few hundred thousand, which interact with each other. Those agents behave based on a few simple rules, which they can modify as they learn. Each agent has access only to local information; it knows what it and its neighbors are doing, but no one agent sees the whole picture. The result of the actions of all of the participating agents produces, in a bottom-up fashion, surprisingly complex and sophisticated adaptive emergent behavior.

A prime example of this is the ant, the bane of homeowners and picnickers alike. Ants illustrate swarm intelligence in which thousands, even millions, of individuals with limited cognitive capabilities and negligible ability to achieve much alone can produce feats of amazing sophistication, such as building a nest, foraging for food, taking care of the brood, allocating labor, and more. Each individual ant within the group follows a limited set of rules to achieve their assigned task. Those that succeed leave a message to others charged with the same task who follow. Although the individual intelligence and capabilities of each ant isn't very great, the collective or swarm intelligence and capabilities are stunning. Researchers have used this model to accurately simulate many systems involving large numbers of independent actors, from stock markets and automobile traffic to epidemics and crime waves.

Software designers and builders can benefit from the insights found in complexity science. A CAS has three main attractive characteristics. First, complex adaptive systems are resilient in the face of failures or unforeseen circumstances. Second, a CAS exhibits homeostasis, the tendency of a system to maintain internal stability. Finally, a CAS can produce extremely sophisticated and complex behavior despite being driven by only a few simple rules. CAS needs just enough process; the resulting semi-chaos fosters adaptability.

Thinking about a 'piece' of software (already our language betrays us into monolithic thinking) as a complex adaptive system might in fact help us architect more resilient, homeostatic, sophisticated software. Early examples may point the way: self-healing, self-configuring systems are found in software for routers that recognize network bottlenecks and steer traffic around them, anti-virus systems that identify and neutralize intruder code, and peer-to-peer networks that store and forward files.

A key challenge in constructing these systems is to enable enough communication between the agents that complex behavior can arise, without unduly constraining the types of communication. As described above, the typical approach in the past has been to overspecify the format, type, and style of communication. To truly liberate the power of emergence in software, we must become smarter about building minimalist standards.

Life itself provides evidence that a very minimal set of standards is sufficient to build complex systems.

Four bases and a handful of amino acids comprise the biological standards used by DNA. Yet, this small set of standards produces the vast variety of life on Earth. Just imagine asking an XML jockey to create a schema sufficient to describe all life forms on Earth. Do you suppose that you would get something as small—and elegant—as DNA? Over the next few years, as we build up increasingly complicated standards for web services description and discovery, we must harken to this lesson. The minimal set of standards necessary to support global distributed computing is undoubtedly smaller, yet more expressive, than current proposals.

Moving to complex adaptive systems as an architecture for software will not be an easy change to make. The very idea that there is no single node coordinating the action is an uncomfortable one, and one with which we have comparatively little experience. And while there is indeed an entire discipline of computer science that deals with distributed systems, it has largely dealt with at most a few dozen instances of highly complex actors, not the hundreds of thousands of simple agents. And even state of the art distributed systems remain extraordinarily difficult to synchronize, coordinate, and manage. To make forward progress, we will have to change many of our notions of computer science education and profession.

Reform the Guild

Creating software for the networked age will require software makers to broaden their tools, education, methodologies, and ultimately the definition of what it means to be a creator of software. Three main forces impel change: first, using complex adaptive systems as an architecture is radically different from current methods. Second, the increasing reach of software in society means that there is more diversity in the types, domains, and requirements of software. University of Illinois professor Ralph Johnson points out that corporate IT support, computer gaming, NASA missions, media entertainment, and others each require different approaches and different skills. Similarly, creating software for single users at their desktop entails a different set of tools and tradeoffs than creating software for, say, millions of Web users. And third, the familiar and persistent problem that, in apparent violation of the laws of economics, there is a continuing—and worsening—shortage of programmers. Overcoming these challenges will require the guild of software creators to reform itself, in the purest sense of the word.

In the past, momentous 'gateway' events transformed the definition of software and ushered in a vastly larger population able to create, manipulate, and understand it. The introduction of assembly, the first high-level languages and compilers, BASIC, spreadsheets and macros, HTML and web programming, each accompanied with an order-of-magnitude increase not only in raw numbers of programmers, but in the diversity of the population. Incidentally, each new stage was also characterized by those already within the guild as not 'real' programming. This doesn't mean that the old kinds of software go away: in the future, there will still be accounting systems, websites, and simulations of protein folding. But the great success of software so far has fed an insatiable appetite for more software, in more fields of life, that do more for us. Moving beyond the current realms of software will require us to change.

As software continues to become more pervasive and adaptable and critically involved in more aspects of life, the ability to create software may become as important as literacy; in fact it may become a fundamental ingredient of literacy itself. Somehow, we must open a new gateway that allows billions of people to be able to create, shape, direct software. This will require completely different conceptions of 'software' and 'programming' than we use today, and will redefine our ideas of the software profession. Author and consultant Alistair Cockburn has observed that our current software tools are the equivalent of a camel-hair brush; we have no programming techniques equivalent to splashing a whole wall with a bucket of paint.

Already, different metaphors are emerging for different types of software creation. The idea of 'software engineering' is now joined by the idea of creating software as a craft involving only a few masters, or as a federation involving a community of tens of thousands. In each of the three metaphors, the role of master, user, and programmer become radically different.

Another approach to reinventing the field is found in Richard Gabriel's Feyerabend Project. Inspired by the work of philosopher and science historian Paul Feyerabend, the project seeks to reinvent the basis of computing by taking seriously broader human concerns than merely computational correctness, such as art, craft, and the basic right and joy of tinkering. These switches would entail enormous implications for software and software creators.

So what might the role of the software guild be in these future? If everyone can create software, does that obviate the need for dedicated professionals? Hardly. Universal literacy hasn't removed the need for professional writers; if anything, the demand for their services is orders of magnitude greater. It is simply that the nature of their work has changed. Similarly, the nature of the work of the software guild must change.

One task that the guild should take up is standard design. Above, we spoke of the increasing importance of setting the standardization level just right; enough to enable emergent behavior but no so much as to inhibit innovation. Software professionals, with their training in distributed system design, may be ideally suited—and positioned—to create and advocate these standards.

Revolt Against the Tyranny of the Turing Machine

The dominant paradigm in software and computation is that of control: absolute control of the machine, of the process being modeled, of the environment, and of communications. Just think of the derivation of the word 'program.' Despite the fact that this idea underlies all of our efforts in computing, it is dangerously wrong. To have any hope of forward progress, software creators must repent from this orthodoxy, and abandon these lies: that a computer and the world in which it operates are deterministic; that an environment can be completely described; and that software infallibility is achievable or even desirable.

As defined by physicist James Clerk Maxwell, the doctrine of determinism asserts that “in every case, without exception, the result is determined by the previous conditions of the subject.” Deterministic systems have driven the creation of software since the start. From Babbage's Analytical Engine to the Turing Machine, our conceptions of computing have been rigidly mechanical. While computing pioneer Alan Turing himself, far ahead of his time, described non-deterministic behavior in his formalisms, the behavior of the standard computer model today is completely determined by the state the machine is in, the input it receives, and its set of instructions. Turing machines have perfectly reliable and infinite memory; they never accidentally change to an undesired state; no malicious entities rewrite the tape when Turing isn't looking. In short, it describes a reliable mechanical process to derive results. Programmers expect the computer will execute a line of code the same way every time and it will produce the same result given the same input. The problem with this, of course, is that it is simply not true for any reasonably-sized system.

The number of different variables in any operating system, application, and environment makes it impossible to reproduce an instance of computer activity exactly. There's an exponentially large number of combinations. To make matters worse, the world isn't digital, it's analog, and we have learned from chaos theory that small, even tiny differences in the initial conditions (as is inevitable with analog values) can have enormous effect later. Whether non-determinism is good or bad doesn't matter; it is the way of the world.

The second great myth of modern computation is completeness. The presumption is that we can somehow completely describe a problem, an environment, a task, or a user's view of the world. We spend enormous effort expanding our views and definitions, hoping that each successive release will be large enough to encompass the world. For any problem domain that involves humans, this is impossible. You can expect the unexpected, but you cannot predict it.

Instead of using test suites to anticipate every eventuality, programmers need to build software that can handle unexpected circumstances. Instead of trying to use old-style techniques to link components into a single application, they must accept that some critical components will inevitably be out of their control and will change their interfaces and standards at inconvenient times. And instead of optimizing for a single context and measures of success, they must allow for runtime changes not merely of the code, but of the very definition of success.

The final myth is that software infallibility is both desirable and achievable. Quality Assurance departments try to produce bug-free software; software products are built to last forever; resources are assumed to be available all of the time. Again, this is a set of unreachable goals. Software, as a complex system (whether or not it is adaptive) simply has too many connections and interdependencies to be absolutely free of bugs.

Instead of striving towards perfection, we should instead embrace imperfection. To paraphrase a famous programmer motto, bugs may be a feature, not a bug. Like the mutations and genetic variation found in biological populations, bugs represent opportunities for adaptation. It may be that the '404' error on the web (indicating a page not found) was a key component to its success; trying to build a system in which you could not address a non-existent page would have made the web far more complicated, restrictive, and unscalable. Our livers are composed of millions of cells performing their liver-duties; we do not die if a single cell dies.

Why must we build software that cannot survive the failure of a single component?

Sometimes, the accumulation of bugs represents an opportunity for death. The problem with the Y2K bug was not that the programmers back in the 1960s were so short-sighted (though they were); it was that our whole system of software usage allowed such programs to remain in place for decades. Organisms die to make way for further generations, but software generally doesn't die, and when it does it often orphans its users and its data. We must allow for death but also be smarter about it.

Killer Apostasy

Already, we see early signs of these new ideas being applied to practical issues. Routers and anti-virus systems exhibit adaptive behavior. CAS concepts are being used to model the behavior of heavy metropolitan traffic. Swarm intelligence approaches are being applied to supply chain management, answering questions like how a system should respond to disruptions in the air transport system that slow the delivery of just-in-time materials. Similarly, these concepts can be used to model pricing in complex, dynamic markets and consumer behavior.

The trick will be to make the leap from creating models based on these concepts to creating software that actually incorporates these concepts as the heart of the very system itself. This means creating software that doesn't just model swarm intelligence or organic process but actually employs it to perform the intended task. This doesn't mean that every old technique goes away, but it does mean that we need to integrate this new tool into our repertoire, and understand better how it fits in with those things that we do know how to do.

The eventual arrival of post-electronic computing, whether based on DNA, photons, neurons, molecules, or quanta, offers the opportunity to finally break from the limitations of the deterministic approach to software. Whatever the device, the software of the future will likely be non-mechanistic and

non-linear at its core. Software creators must accept, even embrace, a level of anarchy if they are to avoid building the brittle, precarious, fragile systems so prevalent today.

This will require new tools, new training and education, new paradigms, and new computers. Indeed, the great challenge of moving to this new world will not be the technical challenge of building new machines, or finding a new killer application, but the rejection of the orthodoxy, an ideology so deeply embedded in our assumptions, language, and literature that we are often unconscious of it.

Martin Luther feared his reform efforts would amount to little more than a fool's song. Perhaps our call to embrace organic computation and liberate ourselves from deterministic Turing machines will turn out as badly as Luther feared. However, according to author Cory Doctorow, “lies about the future inspire people who actually know how to build the future to build it.” In that case, even if it is a fool's song, it still might be what could and should be done.

About the Authors

Geoff Cohen is a Manager at the Cap Gemini Ernst & Young Center for Business Innovation in Cambridge, Massachusetts, where he directs the research on technology, networks, and strategy. His research includes issues in technology adoption, software methodologies, business models, and collaboration technologies. He has also worked for the Congressional Budget Office, Data General, and IBM. He holds a Ph.D. in Computer Science from Duke University and a B.A. from the Woodrow Wilson School of Public and International Affairs at Princeton University.

Alan Radding is an independent writer, researcher, and analyst specializing in business and technology. His writing can be found in leading business, finance, technology, and telecommunications publications and web sites. Mr. Radding also handles a variety of assignments for leading research and consulting firms and major manufacturing and service companies. You can learn more about his work at www.technologywriter.com.