Mob Software: The Erotic Life of Code

An Essay in First Person by

Richard P. Gabriel & Ron Goldman

Let's rock!

I've got good news: That way of hacking you like is going to come back in style.

~

Over the years I've despaired that the ways we've created to build software matches less and less well the ways that people work effectively. More so, I've grown saddened that we're not building the range of software that we could be, that the full expanse of what computing could do—to enhance human life, to foster our creativity and mental and physical comfort, to liberate us from isolation from knowledge, art, literature, and human contact—is left out of our vision. It seems that high-octane capitalism has acted like an acid or a high heat to curdle and coagulate our ways of building software into islands that limit us.

But like survivors, we've managed to make these islands homes. We've found the succulent but bitter fruit that can sustain us, the small encrusted or overfurred creatures we can eat to survive, the slow-moving and muddy streams from which we drink against the urge to spit it out. Being survivors, we can make do with little. But how little like life is such an existence.

~

So gloomy. Bitter and depressing. Deeply conflicted.

The way out requires just one thing from us—a strange, frightful thing—something slant. It is this: Find a way to fight our fear of death.

First, though, the gloom. Feel it.

~

Maybe there aren't too many in this audience old enough to recall the early attempts at putting capsules and rockets into space. What most people remember are the many successful launches in the 1960s and '70s where astronauts first were launched into space but not into orbit; then there was a series of orbital flights, then forays near the moon, then a moon landing, and finally the boringly routine trips gathering rocks and dust, playing golf and goofing off. This is the way to do engineering, many say. Large teams superbly coordinated producing success after success. Software should be like this, they say.

But these folks don't remember these other scenes: One rocket lifted up about a dozen feet, fell over, then accelerated horizontally toward a launch building; another lifted up a few hundred feet and started spinning in a Yeatsian gyre before being blown up remotely; others simply exploded somewhere in the sky. Once I viewed an unnarrated series of these explosions, one after another in nothing less than a ballet of destruction and failure.

NASA began operation on the first of October, 1958. Alan B. Shepard Jr. became the first American in space on the fifth of May, 1961, launched on the tip of a Redstone rocket. On the twenty-first of July, 1961, Gus Grissom became the second American in space, also launched by a Redstone rocket. And on the twentieth of February, 1962, John Glenn became the first American to orbit the earth, launched by an Atlas rocket.

Up until Alan Shepard's launch into space, the Redstone rocket had experienced failures including launchpad and downrange explosions 23 times—the last failure was 2 months before Shepard's flight. My favorite failure took place on the twenty-first of November, 1960, when the Redstone's engines shut off 1 second after ignition: The rocket lifted off a couple of centimeters, and then sat down—with no significant damage. Between Alan Shepard's flight and Gus Grissom's, there was a Redstone failure.

Atlas rockets experienced 36 failures before John Glenn's flight—one occurred the day after Glenn went into orbit.

We like to think we can simply imitate a mature engineering practice and get the same results. As if a Little League baseball team could watch the New York Yankees and say,"hey, let's just imitate them—we'll win all the time!" But success requires fundamentals, and we don't have them.

The highlights of NASA's history represent a false view of even a mature engineering practice. The early days of major American rocketry were filled with failures, some of them disasters. To get to the moon, we needed crashes and endless fiddling. Why do we think that software practice, still in its infancy, can do better?

Software is full of failure, and will be until we can learn how truly to build it.

Fear of failure is fear of death. In fear of failure, we seek order.

The Mob Introduced

The Oxford English Dictionary defines "mob" this way, among others:

> *A multitude or aggregation of persons regarded as not individually important.*
>
> —Oxford English Dictionary, Second Edition, definition 4a, part 2

Let me say it plainly: We know how to produce small portions of software using small development teams—up to 10 or so—but we don't know how to make software any larger except by accident or by rough trial and error. —Because the software we're trying to build is too massive—it is simply too difficult to plan it all out, and we have no idea how to coordinate the number of people it takes. Every piece of software built requires tremendous attention to detail and endless fiddling to get right.

In response to this problem we have clung to fads: structured and object-oriented programming, UML, software patterns, and eXtreme Programming. We grasp for mathematics or engineering to come to our rescue—perhaps even the law: By requiring licenses for our developers maybe we can force improvement in software making.

It just won't happen—it's like those rockets: We simply do not know how to get massive software off the ground without crashing and endless fiddling. But we don't accept that.

The way out of this predicament is this simple: Set up a fairly clear architectural direction, produce a decent first cut at some of the functionality, let loose the source code, and then turn it over to a mob.

The Commerce of the Creative Spirit

Ever compare poetry to newspaper articles? Maybe poetry isn't for everyone, but everyone can see there is a difference—in quality, in beauty, in its appeal to humanity. One is daily news; the other is "news that stays news" (Ezra Pound, "ABC of Reading"). Poetry is constructed, is it not? In fact, aren't all creative acts the result of deliberately envisioned, carefully planned work by single individuals?

This passage by the poet and playwright Harold Pinter is typical of how poets envision:

> *The thing germinated and bred itself. It proceeded according to its own logic. What did I do? I followed the indications. I kept a sharp eye on the clues I found myself dropping. The writing arranged itself with no trouble into dramatic terms. The characters sounded in my ears—it was apparent to me what one would say and what would be the other's response, at any given point. It was apparent to me what they would not, ever, say, whatever one might wish. …*
>
> *When the thing was well cooked I began to form certain conclusions. The point is, however, that by that time the play was now its own world. It was determined by its own engendering image.*
>
> —"The Gift," page 144

...created by the single individual? Writers learn by workshopping. Every night across the country, writers sit in groups of 3 to 20, reading and critiquing each other's work. Not only are these professional writers but amateurs who simply want to improve their diaries. Steven Levy said this in "Hackers":

> *Years earlier, Buckminster Fuller had developed the concept of synergy—the collective power, more than the sum of the parts, that comes of people and/or phenomena working together in a system—and Homebrew* [a Bay Area computer club in the 1970s] *was a textbook example of the concept at work. One person's idea would spark another person into embarking on a large project, and perhaps beginning a company to make a product based on the idea. Or, if someone came up with a clever hack to produce a random number generator on the Altair, he would give out the code so everyone could do it, and by the next meeting someone else would have devised a game that utilized the routine.*
>
> —"Hackers," Steven Levy, page 218

...built as the result of detailed plans? Christopher Alexander said this:

> *Let us start by seeing how the great cathedrals, Chartres and Notre Dame, were made within a pattern language.... There were hundreds of people, each making his part within the whole, working, often for generations. At any given moment there was usually one master builder, who directed the overall layout...but each person in the whole had, in his mind, the same overall language. Each person executed each detail in the same general way, but with minor differences. The master builder did not need to force the design of the details down the builders' throats, because the builders themselves knew enough of the shared pattern language to make the details correctly, with their own individual flair.... We imagine, because of the distorted view of architecture we have learnt, that some great architect created these buildings, with a few marks of the pencil, worked out laboriously at the drawing board.*
>
> *The fact is that Chartres, no less than the simple farmhouse, was built by a group of men, acting within a common pattern language, deeply steeped in it of course. It was not made by "design" at the drawing board.*
>
> —"The Timeless Way of Building," page 216

...but software is different, right? Donald Eastlake said this in 1972 about the creation of the Incompatible Timesharing System at MIT:

> *The ITS system is not the result of a human wave or crash effort. The system has been incrementally developed almost continuously since its inception.*

It is indeed true that large systems are never "finished"…. In general, the ITS system can be said to have been designer implemented and user designed. The problem of unrealistic software design is greatly diminished when the designer is the implementor. The implementors' ease in programming and pride in the result is increased when he, in an essential sense, is the designer. Features are less likely to turn out to be of low utility if users are their designers and they are less likely to be difficult to use if their designers are their users.

—Donald Eastlake, 1972, quoted in "Hackers," page 127

The Swarm

In "The Lives of a Cell," Lewis Thomas wrote about termites building arches:

Termites are even more extraordinary in the way they seem to accumulate intelligence as they gather together. Two or three termites in a chamber will begin to pick up pellets and move them from place to place, but nothing comes of it; nothing is built. As more join in, they seem to reach a critical mass, a quorum, and the thinking begins. They place pellets atop pellets, then throw up columns and beautiful, curving, symmetrical arches, and the crystalline architecture of vaulted chambers is created. It is not known how they communicate with each other, how the chains of termites building one column know when to turn toward the crew of the adjacent column, or how, when the time comes, they manage the flawless joining of the arches. The stimuli that set them off at the outset, building collectively instead of shifting things about, may be pheromones released when they reach committee size. They react as if alarmed. They become agitated, excited, and then they begin working like artists.

— "The Lives of a Cell," Lewis Thomas, (New York: Viking. 99 1974), p. 13.

One of the remarkable discoveries of recent times is that complex behavior by a group of individuals requires only that each individual follow simple rules, and the collective behavior of the group can display facets and filigrees nowhere apparent in those rules. Here are three simple rules:

- Each individual shall steer toward the average position of its neighbors.
- Each individual shall adjust its speed to match its neighbors.
- Each individual shall endeavor to not bump into anything.

With a group of individuals each following these rules, we observe the simple but beautiful and graceful flocking patterns of birds and schools of fish.

Simulation is the main tool of complexity investigation. One of the better known simulations is "Sugarscape," which became noted over the past 5 years by revealing that complex societal behavior could emerge from a small number of individual behavioral rules alongside simple genetic characteristics.

This is the setup: a simple 2-dimensional grid, a population of individuals initially spread randomly over the squares. Each square can be occupied by only one individual. Some squares contain sugar, which the individual can collect and consume. In the initial setup, an individual can move and see a certain distance. Movement costs energy regulated by metabolism. An individual scans the horizon for sugar and goes to the square with the most of it while avoiding squares with other individuals. Individuals can hoard sugar. Each cell in the grid has a sugar capacity, and sugar can "grow back" at some rate after it is harvested.

If we place two large piles of sugar in the grid, the population splits in two and feeds as if each pile were covered by a swarm of ants. If we introduce seasons during which the density of sugar moves according to time, we'll see the population migrate—simply as a result of the simple rules and their interaction through individuals with the environment.

Of course, because not all individuals are the same—they are created with a distribution of traits such as metabolism rate, visual acuity, and collection capacity—some fare less well than others. Some become fat and live forever; others starve, having been born in sugarfree badlands.

The next step is to add some simple mechanisms for evolution and social networking: mating and genetic exchange, meme exchange, actor status, affinity for similar agents, repulsion from different ones, and conforming. Each of these is expressed as simple, local rules. The general principle for complexity design is this:

Think locally, act locally.

When inheritance is added—the ability of a parent to pass on its sugar hoard to offspring—visual acuity decreases and the gap between the rich and the poor widens dramatically.

Cultural transmission and diffusion behaviors emerge. By creating varieties of sugar, economic behavior can be studied. Fads, norms, and fashions emerge. Isolated areas become closed societies. Adding combat-related rules creates wars and power struggles.

Is this eerie? Silly? Important research? Or just so-what?

Such phenomena point to an interesting locale in complex systems—the border between order and chaos. Chaos is unpredictability: Combinations that might have lasting value or interest don't last—the energy for change is too high.

Order is total predictability: The only combinations that exist are the ones that always have—the energy for stability is too high.

Stuart Kauffman, formerly of the Santa Fe Institute, wrote:

> *It is a lovely hypothesis, with considerable supporting data, that genomic systems lie in the ordered regime near the phase transition to chaos. Were such systems too deeply into the frozen ordered regime, they would be too rigid to coordinate the complex sequences of genetic activities necessary for development. Were they too far into the gaseous chaotic regime, they would not be orderly enough.*
>
> —"At Home in the Universe," page 26

and

> *...cell networks achieve both stability and flexibility...by achieving a kind of poised state balanced on the edge of chaos.*
>
> —"At Home in the Universe," page 86

Between order and chaos, interesting and unexpected combinations come about and last long enough to have repercussions. Trends can be observed. Patterns emerge. The connection to poetry is remarkable: One way to look at poetry is that release—the wild imagination of Whitman and Garcia Lorca—is reined in by restraint: requirements of form, grammar, sentence-making, echoes, rhyme, rhythm. Without release there could be nothing worth reading; the erotic pleasure of pure meandering would be unapproached. Without restraint there cannot be sense enough to make the journey worth taking. Poetry is at the margins of understanding, in the fractures of our reality, in the space between order and chaos.

Listen for a minute to Federico Garcia Lorca at the end of the poem "Scream to Rome," written in New York around 1930:

> Meanwhile, meanwhile, oh!, meanwhile,
> the blacks that take out the spittoons,
> the boys that tremble beneath the pale terror of the directors,
> the women drowned in mineral oils,
> the crowd of hammer, violin, or cloud,
> will scream although their brains may blow out on the wall,
> will scream in front of the domes,
> will scream maddened by fire,
> will scream maddened by snow,

will scream with their heads full of excrement,
will scream like all the nights together,
will scream with a voice so torn
that the cities tremble like little girls
and the cities of oil and music break,
because we want our daily bread,
flower of the alder and threshed tenderness,
because we want to be fulfilled the will of the Earth
that gives its fruits for all.

This is a poem of duende—neither angel who dazzles but hovers above nor muse who whispers and dictates. Duende is a force that dares us—when we empty ourselves, approach ourselves—to see truly; taunts us into being serious; battles order. All that has black sounds has duende. Garcia Lorca says:

> *The duende…is a power, not a work. It is a struggle, not a thought. I have heard an old maestro of the guitar say, "the duende is not in the throat; the duende climbs up through the soles of the feet." Meaning this: It is not a question of ability, but of true, living style, of blood, of the most ancient culture, of spontaneous creation.*
>
> —"In Search of Duende," Garcia Lorca, page 49

Garcia Lorca says the duende

> *will not approach at all if he does not see the possibility of death*
>
> —"Poet in New York," Garcia Lorca, page 162

and

> *we only know that he burns the blood like a poultice of broken glass, that he exhausts, that he rejects all the sweet geometry we have learned, that he smashes style.*
>
> —"In Search of Duende," Garcia Lorca, page 51

Fear of failure is the fear of death, the fear of duende. Christopher Alexander said,

> *All of the Japanese arts recognize that, finally, you have to meet the fear of death in order to do anything—landscape painting, flower arrangements, and so on."*
>
> —"Christopher Alexander: The Search for a New Paradigm in Architecture," Stephen Grabow, page 86

Allen Ginsberg said that to be a poet you must abandon fear of failure by

> *...abandoning the glory of poetry and just settling down in the muck of your own mind...*
>
> —"The Gift," page 145

When you do this—when you do battle with the duende—you might find a gift, a gift of talent or insight that will make you think,"Did I say that?" —"Did I do that?" Gifts like this are worth cultivating, even in the software world. Lewis Hyde wrote this in "The Gift":

> *So long as the gift is not withheld, the creative spirit will remain a stranger to the economics of scarcity. Salmon, forest birds, poetry, symphonies, or Kula shells, the gift is not used up in use.*
>
> —"The Gift," page 146

Duende, poetry, and perhaps life itself and life in our works of artifice are denizens of the boundary between order and chaos. When we look too deeply into chaos for inspiration, we find only nonsense; when we look too deeply into order for completion and closure, we find only the botfly of boredom and the disease he carries: failure and deathlike morphology.

Such an interesting place is the border between order and chaos that Dee Hock coined a word for it: ***chaord***. His work from the mid-1960s has focused on how to replace outworn hierarchical, command-and-control organizations and what to replace them with. Command-and-control systems are based on the need for control, predictability, and order, things that Stuart Kauffman says have no real place in biologically based systems. What is needed is the vigor at the border of chaos and order.

Dee Hock reasoned that if organizations based on biological principles were to replace traditional command-and-control systems, then the basis for the self-organization and emergent behavior of a group of people would have to be the same as what forms the similar basis in life: a genetic code. In the Sugarscape simulation the genetic code is represented by local rules of behavior. A rule of behavior is like a principle. If a group shares the same principles (Dee

Hock figured) then they would be able to work in emerging concert without the need for command and control.

But nothing can be built without a purpose, some idea of what to build. Dee Hock was trying to construct a system that would accomplish the electronic exchange of value—that is, money. To build this, he needed to build an electronic infrastructure—a network and accompanying software—and an organizational infrastructure that would carry out the exchange of money the electronic transfers promised.

Because Dee Hock took seriously the motto, "think locally, act locally," he intuited that the best organizational concept for such an undertaking would move decisions as close as possible to those who feel their effects.

The result was known as Visa International, and it still exists today, though not entirely in the same form.

Dee Hock's ideas should sound very familiar. Building a cathedral is the shared purpose, and the builders—from the master builder down to the most insignificant member of the mob engaged in its construction—knew the elements: Nave, aisles, transepts, the great rose window in the west end, and the chapels round the east end. The builders shared a set of principles—the common pattern language. Alexander said:

> *But still the power and beauty of the great cathedrals came mainly from the language which the master builder and his builders shared. The language was so coherent that anyone who understood the language well and devoted his whole life to the building of a single building, working at it slowly, piecemeal, shaping all the parts within the common pattern language, would be able to make a great work of art.*
>
> *The building grew slowly, magnificently, from the impact of the common pattern language from which it was made, guiding its individual parts, and the acts which created them, just as the genes inside the flower's seed guide and then generate the flower.*
>
> —"The Timeless Way of Building," page 216

The Changing Face of Software

Founding assumptions can echo for decades. In software we have suffered especially from such echoes. From the earliest days of software—from the mid-1950s—a particular view has prevailed of what software is and why it is created that now or soon will make doing our work intolerable.

Computing practice was first developed as a branch of computability theory based on a particularly inhospitable set of mathematical constructs: group and

semi-group decidability, Thué and other transformational systems, recursive function theory, lambda calculus, and Turing machines, to name some. In mathematical terms, these constructs are equally expressive, universality being the touchstone of expressiveness. With this set of blinders firmly in place, there is no difference between any of these models, and the same logic is applied with equal blindness to programming languages.

Early computing practices evolved under the assumption that the only uses for computers were military, scientific, and engineering computation—along with a small need for building tools to support such activities. We can characterize these computational loads as numeric, and the resulting computational requirements as Fortran-like. Early attempts to expand the realm of computing—such as John McCarthy's valiant attempt with Lisp and artificial intelligence, Kristen Nygaard's similarly pioneering attempt with Simula, Doug Englebart's intriguing attempt to expand our usage options, Alan Kay's later attempt with Smalltalk, and Alain Colmerauer's attempt with Prolog—were rebuked.

The very architecture of almost every computer today is designed to optimize the performance of Fortran programs and its operating-system-level sister, C. Further, attempts to consider neuron-based, genetic, or other types of nonstandard computational models were soundly rejected, snubbed, and even ridiculed until the late 1980s.

Programming languages have hardly shown one scintilla of difference from the designs made in the first few years of computing. All significant programming languages are expressively, conceptually, and aesthetically equivalent to Fortran and assembly language.

In fact, what it means to program is hopelessly outdated. Early models persist. In 1960, one generally coded up a single program, linked its parts together in a monolithic, monochronic puddle of behavior and data—everything brought together in one place at one time, everything made consistent by tools that ensure such consistency.

Software development methodologies evolved under this regime along with a mythical belief in master planning. Such beliefs were rooted in an elementary-school-level fiction that great masterpieces were planned, or arose as a by-product of physicists shovelling menial and rote coding tasks to their inferiors in the computing department. Master planning feeds off the desire for order, a desire born of our fear of failure, our fear of death.

~

How can we tolerate these indignities? The frustratingly simple answer is the universal nature of programming languages—the fact that one can program in any one of them what can be programmed in any other. We are able

to clever our way out of any programming box. The more difficult the task, the more pride we can take in the accomplishment.

We are loathe through either our natures or our upbringing to admit that the programming tasks we are called upon to do are too hard with available tools or that better tools would improve our lives. We cannot easily see that we are boxed into a corner by the assumptions of our ancestors—as correct as those assumptions were or seemed to be at the time.

~

The reality today of programming's needs are so far off from that early view that we really should be belly-laughing.

Until recently, the model of software creation and use held that programmers put together software behind closed doors and people used it only after copies had been distributed—on floppy disks and CDs, or by downloading. This view of software distribution is being overturned by the concept of ***performing software***. In this notion, software is created whichever way makes sense, but its only executing incarnation is on a single or a few machines connected to the Net, and its users connect to that machine or those machines to use it. On the horizon is an obvious tipping point beyond which there is just one, immense distributed system made up of a large number of performing programs that interact with each other.

Such future distributed systems will have been built by vastly diverse, dispersed, and different-minded people—and massive numbers of them. Each such system will never go down, cannot be recompiled from scratch, and can never be in total version coherence.

Even in monolithic systems we find the same thing. For every major operating system, there are people who absolutely demand and rationally require the newest backwardly incompatible features, and others who absolutely and rationally require the oldest features to remain as they are forever.

A simple example can be found in the Java™ language and Jini™ Technology. Jini is a simple distributed computing model based on the Java programming language. Among other things, it was intended as a model for services—small bits of functionality—to discover each other dynamically and create a network of interoperating program parts. These services could be housed within devices—physically separate computing platforms as far as Jini is concerned. Because such a model is new, the definition of each service is likely to require a maturation period, in which an initially immature service is defined and released, with perhaps numerous improvements subsequently being released. Service definitions are specified by Java interfaces: API's, in other words.

The natural way for a community of service developers to work is to start with a provisional service definition and move through a series of improvements,

culminating, perhaps, in a ratified service definition residing in the `net.jini` package. Jini quite cleverly handles this in some cases because the implementation of the service comes along with the device, but the Java interfaces may not match very well. Perhaps they will not obey even the Java binary compatibility rules. There are no mechanisms within Java to handle mismatched interfaces. There are not even any mechanisms to migrate an interface from one package to another.

This is a problem of interface evolution and simple renaming. If it's true that software systems must be living systems, our views on the static construction of programs need to be updated.

Indeed, there are some programming languages that can handle this form of interface evolution, modeled on similar mechanisms in the database world (so-called ***schema evolution*** mechanisms), but these languages are not looked on favorably.

One of the languages that supports schema evolution for classes operates by allowing developers to specify repair code that scampers about when an unrepaired instance is found and repairs it. Even though the designers of this system didn't think of it this way, this is a clearly biological approach to the problem: a swarm of small repair agents fixing up code on the fly to make it work when the context of its operation has changed.

Every single programming language we have—including Java—is predicated on the physicists' model of figure it out, code it up, compile it, run it, throw it away. All our computing education is based on this, all our methodologies, all our languages, environments, tools, attitudes, mathematics, prejudices, and principles are based as solidly and firmly on this rock as Gibraltar is based on the stone it rests on. Many if not most development headaches come from precisely this model—headaches that programming language theorists ridicule. Every attempt to define a language to handle this or a computing device to make it simpler is met with laughter and rejection. And all this in the face of the fact that the reality of programming must be based on precisely the opposite assumptions: Everything changes, every version is necessary, evolution happens.

Resource Limits

The history of computing is rife with resource limitations—to the extent that there was an attempt at a theory in the 1970s to explain intelligence and other human-level performances using "resource-limited computation."

We have labored under three types of resource limitations:

- Until the 1990s we did not routinely use computers with much more than a few dozen mips and 4–8 megabytes of RAM.
- We typically run development projects undermanned by a factor of 2 or 3 and schedules about 50% too short.
- We typically underspend per unit time by a factor of 2 or 3.

Because of this we've managed to build only very small pieces of software, and even those are horribly error-ridden. Because of the apparent need to constantly innovate—or obsolete—hardware and to renew ourselves creatively, and because development has been done primarily in a landscape riddled by many isolated islands of similar or identical activities, we have, over the last 40–50 years, written the same code over and over. How many versions of Unix do we need? Do we really need the few dozen that we have? How about a text editor more advanced than Emacs? Are Word, Excel, and Powerpoint really the best possible programs in those categories? We've built these programs many times, and we don't get particularly better at them.

We have gotten OK at building cathedrals—only we are building the same dozen or so over and over again. Our patterns are too limited and focused on efficiency, our vision outworn; sources of new language have dried up, and it's as if we are caught in a delirium of mere rephrasings.

However, the commercial software community has developed one particular response to resource limitation: a fevered, workaholic approach to software development—error prone, hectic, family-destroying, health-degrading, night-haunting. If you are undermanned by a factor of 2, add a second 8-hour workday per physical day. If you are operating under a schedule 50% too short, add in another 32 hours per week by working weekends. Then pray for luck or push back on features and quality.

Scarcity breeds a commodity or exchange economy. Until almost 1980, there were essentially no markets for software. Before 1980, most computers were owned by companies and used for "large tasks." When computers were commoditized, the resource limitations inherent in software development became an opportunity for exploitation, and any relief to those limitations meant less wealth to go around. Draw your own conclusions.

Order and Modular Parts

Programming language designers made one important breakthrough in being able to program effectively given insane resource limitations: the modular part—call it subroutine, procedure, function, method, or class. The idea has two parts:

- Create a piece of code that performs some computation.
- Create a way to run the code in several contexts.

The first of these is the code that will be used over and over. Typically, what this code does depends on the local context. The second of these serves two purposes: One is to provide a way to inform the modular part about the local context; and the other is to provide a sort of description of the modular part—this is conveyed primarily by the modular part's name, but its arguments can sometimes provide additional descriptions, particularly for methods.

Modular parts seem like the ideal of order: Construction is uniform, expectations are identical, fear is quashed, calm reigns—because things rarely change in a fixed landscape. As I implied before: Why are we continually writing the same software over and over?

The Ariane 5 disaster says a lot about software development—recall, it was the European Space Agency rocket that self-destructed after its engine nozzles all swiveled into an extreme position, causing the rocket to veer abruptly, rupturing the links between the booster and the core stage, which caused the rocket to self-destruct.

The failure has been analyzed thoroughly by both the ESA and software professionals, with most of the discussion focusing on uncaught exceptions, best-effort results, and the foolishness of simply shutting down a system as a reasonable approach to failure. But the failure was caused by software adopted without adaptation from the Ariane 4. The decision was made to reuse the code by Ariane 5 software developers and designers who had too much confidence in code that had proved successful many times in the past.

I quote from the report concerning a special feature which continued to run a calibration routine 50 seconds into the flight:

> *The same requirement does not apply to Ariane 5, which has a different preparation sequence, and it was maintained for commonality reasons, presumably based on the view that, unless proven necessary, it was not wise to make changes in software which worked well on Ariane 4.*
>
> —Paris, 19 July 1996 "ARIANE 5: Flight 501 Failure Report by the Inquiry Board,"
> The Chairman of the Board: Prof. J. L. LIONS

The failure, therefore, had something to do with the use of modular parts: failing to look specifically at the requirements for the new system and assuming that a similar module for a different system—which was well-tested under real conditions—should be trusted unless proven otherwise.

If we look at ourselves literally as systems we notice soon enough that although we are each made of very similar parts, we are not made of modular parts. Arms look like modular parts, but they aren't. Each is custom grown for the person who hosts it. Within each person's genetic code is the knowledge of how to grow an arm in place, custom fit to the body being created concurrently.

Moreover, we already seem to know this in some aspects of mainstream programming. A database may seem like a modular part, but it isn't. A database is created by taking off-the-shelf genetic coding in the form of a database system. Then we reckon up schemata defining how the data will be laid out, and we add data to made a database.

Let's listen to Christopher Alexander on the topic of modular parts:

> *The details of a building cannot be made alive when they are made from modular parts.... And for the same reason, the details of a building cannot be made alive when they are drawn at a drawing board.*
>
> —"The Timeless Way of Building," pp. 460–461

Modular parts: Maybe they aren't so wonderful.

~

The real problem with modular parts is that we took a good idea—modularity—and mixed it up with reuse. Modularity is about separation: When we worry about a small set of related things, we locate them in the same place. This is how thousands of programmers can work on the same source code and make progress. We get in trouble when we try to use that small set of related things in lots of places without preparing or repairing them.

Who Can Be Programmers?

Computing practice and theory is based on the very hidden and hard-to-reveal assumption that engineers, mathematicians, and computer scientists are the only ones who will write a program or contribute to a software system. When the computing age began, no one but scientists and the military even remotely conceived of the utility of programs and programming. Today, almost every business and human pursuit is built on computing and digital technology. Artists, craftspeople, writers, fishermen, farmers, tightrope walkers, bankers, children, carpenters, singers, dentists, and animals depend on computing, and most of the people I mentioned want to have a say in how such software works, looks, and behaves. Many of them would program if it were possible. The current situation might feel fine to some of you, but suppose all computing were based

on the needs of tightrope walkers? Hard to imagine? What we've created was hard for them to imagine.

Literature of Code and Teaching

Fast-lane capitalism has created a nightmare scenario in which it is literally impossible to teach and develop extraordinary software designers, architects, and builders. The effect of ownership imperatives has caused there to be no body of software as literature. It is as if all writers had their own private companies and only people in the Melville company could read "Moby-Dick" and only those in Hemingway's could read "The Sun Also Rises." Can you imagine developing a rich literature under these circumstances? Under such conditions, there could be neither a curriculum in literature nor a way of teaching writing. And we expect people to learn to program in this exact context?

When software became merchandise, the opportunity vanished of teaching software development as a craft and as artistry. The literature became frozen. It's extremely rare today to stumble across someone who is familiar with the same source code as you are. If all remnants of literature disappeared, you'd expect that eventually all respect for it—as an art form, as a craft, as an activity worthy of human attention—would disappear. And so we've seen with software: The focus is on architecture, specifications, design documents, and graphical design languages. Code as code is looked down on: The lowest rank in the software development chain is "coder"—right alongside QA drone and doc writer.

We find little or no code education. Typically a student will write a few dozen small programs over the course of an undergraduate education, and perhaps he or she will work on a larger program—such as a compiler—with a small group of students. At that point, the student is ready to become a professional programmer. Little or no mentoring is done after that, and if the student becomes a proficient programmer, it is due to luck and hard work, in that order.

Birth of Mob Software

A few years ago, the open source community posed a significant question—a question which has been drowned out in debates over licensing and the politics of open source. The question was:

What if what once was scarce is now abundant?

Hidden in the premises of open source is the belief that perhaps it's acceptable to waste programmer time. Moreover, there are no time limits to open-source

projects. With lots of people looking at the source code, it's easy to see how the quality of the resulting code could be a little higher. And with hackers always desperate to prove themselves wizards, it's easy to see how "customer support" could be a little better.

Because the open source proposition asked the crucial first question, I include it in what I am calling "mob software," but mob software goes way beyond what open source is up to today. What I'm talking about is the kind of swarming activity we see exhibited by social insects, a kind of semi-chaotic self-organizing behavior in which numerous small acts of repair with a common goal in mind can lead to quickly built, complex, and massive creations. Mob software is divergent—it explores the space of possibilities. It includes users during design. It is highly incremental, focusing on repair rather than master planning. Mob software takes coding out of the closet and makes code literature. It is based on gifts. Mob artifacts include massive software, built by the multitudes. It never goes down, it is never totally stable but is stable in the aggregate. It is flexible in the extreme. Anyone can add to it. Mob-software artifacts adapt, evolve almost. It is living while all the software we have now—almost all of it—is dead-dead-dead. Mob software is the technical software community finally coming to grips with the duende.

Open source is only some of these things, but it provides our first view into mob software, so let's talk about open source for a moment.

By now we know about quite a few mostly successful open-source—or baby mob software—projects: Linux—which has won a number of awards for customer support and artistry—Mozilla, Jikes, Emacs, GCC, Apache, Perl, Python, sendmail, and BSD.

The Early History of Open Source

Sadly? Happily? Oddly?—well, open source and the mob approach is just not a new idea. A great deal of software development before the 1980s took place in conditions similar to those propounded by the open source folks. If your organization owned a computer, you had a copy of the source code for many of its systems. Back in the 1960s and '70s, PhD dissertations in computer science—and especially in artificial intelligence—contained the source code for the program described in the dissertation. In 1962, the Lisp 1.5 Programmer's Manual contained the mysterious, almost gnostic, Lisp code that implemented Lisp. Though mostly obvious to mathematicians, this Lisp in Lisp took several decades for computer practitioners to get their heads around.

Throughout the 1960s and '70s, and even into the 1980s, it was relatively common to exchange source code. The nascent Internet—called at the time,

the Arpanet—was created in order to make computing facilities available to researchers around the world, and to exchange files including source code. One of the first net-based "open source" projects was Maclisp—a PDP-10–based simple Lisp that became one of the dominant Lisp dialects in the 1970s. Originally developed at MIT under ITS, it was ported to several other popular PDP-10 operating systems by a group of about 10 volunteers who donated their time and worked on the code partly at their own locations and partly over the Arpanet. Developers who earned the respect of the principal maintainers of Maclisp at MIT were given the authority to edit the sources on MIT's computers and to build Maclisp there.

Similar activities took place: Languages, environments, planning systems, natural-language understanding systems, game-playing programs, games, text editors, compilers, graphics programs, window systems, typesetting systems—just about anything you can think of was written this way, though on a smaller scale because there were hundreds of computers on the Arpanet, not the nearly 100 million we have on the Internet today.

In the 1980s, it became apparent there was money to be made by writing and selling software, and that a premium might be had for being clever about it. Although software patents were first tried about 10 years earlier, they became increasingly popular during the 1980s until the 1990s, when the intellectual property grab came to resemble the Oklahoma land rush.

At that point, except for academic projects and the activities of the Free Software Foundation under the eclectic oversight of Richard Stallman, software source code essentially disappeared.

The Dictionary

Mob development—the creation of software by a loosely coupled group of volunteers—seems a thoroughly contemporary phenomenon, based on the free outlook of the 1960s, a kind of fallout of free love and hippiedom—I mean, just look at the proponents of this approach: anti-establishment Marxists who believe in free software, who call themselves "hackers"; people with scruffy hair, who don't exactly work for a living, people who attend Star Trek conventions and think science fiction is both high literature and true science.

However, the idea is a bit older. On Guy Fawkes Day, 1857, Richard Chenevix Trench, addressing the Philological Society, proposed to produce a new, complete English dictionary based on finding the earliest occurrences of each of the English words ever in printed use. That is, the dictionary would be constructed by reading every book ever written and noting down exactly where in each book a significant use of every word occurred; these citations would be used to write definitions and

short histories of the words' uses. In order to do this, Trench proposed enlisting the volunteer assistance of individuals throughout English-speaking countries by advertising for their assistance.

Over a period of 70 years, many hundreds of people sent in over 6 million slips with words and their interesting occurrences in thousands of books. This resulted in the Oxford English Dictionary, the ultimate authority on English, with 300,000 words, about 2.5 million citations, 8.3 citations per entry, and in 20 volumes.

Compare this with the best effort by an individual—Samuel Johnson—who, over a 9-year period, using the same methodology and a handful of assistants called ***amanuenses***, produced a 2-volume dictionary with about 40, 000 words and in most cases 1 citation per entry. As we look at these two works, Johnson's dictionary is a monument to individual effort and a work of art, revealing as much about Johnson as about the language he perceived around him, while the OED is the standard benchmark for dictionaries, the final arbiter of meaning and etymology.

Why Mob Software Works

Mob software works by involving a multitude of developers who are passionate about the software being constructed because it is important for them. Mob software projects do not generally have deadlines, but efficiency is gained by more eyes, hands, and minds being applied to the work—in general, fewer major releases are required to get to the same level of functionality and quality. To get an idea, whereas a large software project by conventional standards might have 250–500 developers, a mob software project can have thousands or tens of thousands of developers who are not only actively working on the software but using it every day under a variety of conditions. Moreover, these developers span a range of talents almost no employer could afford to hire. Further, these developers typically include some individuals who have remarkable abilities along with astounding disabilities—they might not even be employable.

Among the main volunteers to the OED project was Dr. William C. Minor, a brilliant etymological worker who, during his entire 30 or so year involvement with the OED, happened to be locked up in an asylum for the criminally insane, where he suffered hallucinations that kept him awake all night most nights and where he eventually mutilated himself in the most horrific manner. Once incarcerated, he was never entirely outside immediate and confining care. Yet he was acknowledged as one of the most important contributors to the project.

Monolithic Development

Some might think that my claim that we don't know how to do massive software is exaggerated, that we do know and it's called SEI Level 5, the highest maturity rating. And the story that explains how things should work is not the one about the OED but the one about space shuttle software.

Lockheed Martin Corporation employs a Level 5 software group that puts together the shuttle software. This software seems large, and it is written by a deliberate process. Let's look at this.

The software is not very big—only 420,000 lines of code. This is not massive software—it's off by a factor of 10 or 100. The software is monolithic, an unrealistic model for future systems. The software was developed over a 20-year period by a group of 260 people. These numbers imply that each line of code has had 25 person-hours of attention, and that each person has been responsible for a little over 1600 lines of code—25 pages. $700,000,000 was spent to develop this code, or about $1700 per line.

Further, the software had been completely specified in pseudo-code agreed to beforehand. The task of a programmer in this regime is to translate pseudocode to actual code. But, all this work does buy NASA only 1 error in each of the last three releases of the code. Because we cannot afford to have deaths in the space program, the cost and effort are worth it. The cost amounts to making each subroutine a career-long research project. Twenty years, 8 hours a day, studying and eliminating errors from 1600 lines of code.

The mob-software theory is that this project needed 26,000 programmers, not 260. The job could have taken less than a year, probably with better quality, and a lot cheaper. Better quality because the range of talent that could have been brought to bear would have been tremendously greater, and cheaper because most of the work would have been a gift. (Motivating volunteers who cannot be users takes a culture of gift-giving we don't have yet.)

We cannot afford to inch our way up the learning curve, 1600 lines at a time. We need to get things done fast while doing some learning along the way.

Gift and Commodity Economies

To understand what the open source and mob software movements are trying to do, it helps to make a distinction between a ***commodity economy***, to which we are accustomed in a capitalist society, and a ***gift economy***. In a gift economy, gifts are exchanged, forming a bond based on mutual obligation: In the simplest form of gift exchange, when one person gives a gift to another, the receiver becomes obligated to the giver, but not in a purely mercenary way—rather, the recipient

becomes very much like a member of the giver's family where mutual obligations are many, varied, and long lasting. More sophisticated forms involve more than two parties—in fact the cosmos may become involved. A person may give a gift with the realistic expectation that someday a gift of equal or greater use value will be received, or that the recipient will pass on a further gift. Sacrifices and many religious ceremonies are gift-economy based. In an open-source project, the gift of source code is reciprocated by suggestions, bug reports, debugging, hard work, praise, and more source code.

Gift economies are embedded within non-economic institutions like kinship, marriage, hospitality, artistic patronage, and ritual friendship. The bond is so like flesh and blood that the Greek gift economy persisted alongside a vigorously growing commodity economy for several centuries.

A gift has both economic and spiritual content. It is personal. In giving a gift the goal is to become as empty as possible. In "Gift and Commodity in Archaic Greece," Ian Morris wrote the following:

> *The aim of the gift economy is accumulation for de-accumulation; the gift economy is above all a debt economy, where the actors strive to maximize outgoings. The system can be described as one of "alternating disequilibrium" where the aim is to never have debts "paid off" but to preserve a situation of personal indebtedness.*
>
> —"Gift and Commodity in Archaic Greece," Ian Morris

In a typical gift exchange situation, a patron would provide food and shelter to a poet in exchange for poems about the patron. If you look at this exchange closely in the terms in which it was made, you can see it as a trade of life sustenance for memory after death. Moreover, each would enjoy the company of the other at the dinner table, and in many cases the wit and eloquence of the poet would be a further gift to the patron and by the patron to his guests.

Notice that in this hypothetical exchange, the gifts are tangible and their value resides in how they can be used: Food is used for sustenance and a poem is used to memorialize the patron in speech before death and in script after. The gifts are thus inalienable—their value cannot be separated from their identity.

In a commodity economy, the value of an item is abstracted into some other sort of object whose intrinsic value is unrelated to its "purchasing power." For gold coins this characterization makes less sense than it does for paper money, which is the ultimate abstraction: Paper money is completely real and yet unrelated to what it abstracts—so much so that it is possible through the medium of paper money to make a completely fair trade between otherwise

incommensurate things, such as 500 cotton candy cones for one handgun through the alienable intermediary of $500.

In a commodity economy, money is an alienable object exchanged by independent transactors. In such an economy there is no need for the poet to sit at the patron's dinner table, and in fact, the poet may have many anonymous and alien patrons who find the work fulfilling or not. More than that, the poems the patron receives provide nothing personal to the patron at all, and the term "patron" inherently makes no sense whatsoever in this case—alien patrons are merely donors.

Most importantly, the commodity economy—let's call it capitalism—depends on scarcity. Its most famous law is that of "diminishing returns," whose working requires a fixed supply. Scarcity of material or scarcity of competitors makes high profit margins. It works through competition.

The gift economy is an economy of abundance—the gifts exchanged are inexhaustible, consisting of ritualized friendship and hospitality.

The gift and commodity economies have co-existed for millennia, generally with the most intimate relationships governed by gift, not money. Tribes based purely on gift economies will barter and exchange money for goods with outsiders. A healthy Western family operates on a gift economy. For mob software to succeed in communities that include corporations, the source-code gift-based economy needs to thrive alongside the commodity economy hovering on its boundaries.

The following quote illustrates the difficulty of co-existence as observed in 6th century BC Greece. In it, the Greek word ***xenia*** means "a bond of solidarity manifesting itself in an exchange of goods and services between individuals originating from separate social units." (Gabriel Herman, quoted in "Economy of the Unlost," page 13):

> *For whereas money is concerned to change the status quo, gifts aim to sustain it. The profound conservatism of a gift economy secures its own continuance and moral prestige in two ways: First, by derogation of all that is not gift. We can see a deep distrust of money, trade, profit, commerce, and commercial persons pervading Greek socioeconomic attitudes from Homer's time through Aristotle's. "Commodity exchange was not an acceptable activity for a Greek."* ["Gift and Commodity in Archaic Greece," Ian Morris]*...At the same time, a gift economy likes to project its functions onto the cosmos...as if the rules of xenia represent the way things are for gods and men.*
>
> —"Economy of the Unlost," Anne Carson

In the gift economy, the most empty individual is the wealthiest, being the most likely to receive gifts, while in the commodity economy, the richest individual is the wealthiest, being the most likely to receive further riches. Trying to reconcile these two economies today will be tough because the gift economy must encroach on the commodity economy, not the other way around as it has always been.

Listen to Lewis Hyde:

> *...the exploitation of the arts which we find in the twentieth century is without precedent. The particular manner in which radio, television, the movies, and the recording industry have commercialized song and drama is wholly new.... The more we allow such commodity art to define and control our gifts, the less gifted we will become, as individuals and as a society. The true commerce of art is a gift exchange, and where that commerce can proceed on its own terms we shall be heirs to the fruits of gift exchange:...to a creative spirit whose fertility is not exhausted in use, to the sense of plenitude which is that mark of all erotic exchange, to a storehouse of works that can serve as agents of transformation, and to a sense of an inhabitable world—an awareness...of our solidarity with whatever we take to be the source of our gifts, be it the community..., nature, or the gods. But none of these fruits will come to us where we have converted our arts to pure commercial enterprises.*
>
> —"The Gift," pp 158–159

Xenia involves emptying one's self, and along with this emptiness comes the duende and with it risk-taking, creation, diversity. The Chilean poet, Pablo Neruda, connected xenia and the duende in this story from his childhood in the southern frontier of Chile:

> *I looked through the hole* [in the fence] *and saw a landscape like that behind our house, uncared for and wild. I moved back a few steps because I sensed vaguely that something was about to happen. All of a sudden a hand appeared—a tiny hand of a boy about my own age. By the time I came close again, the hand was gone, and in its place there was a marvellous white toy sheep.... I went into the house and brought out a treasure of my own: a pine cone, opened, full of odor and resin, which I adored. I set it down in the same spot and went off with the sheep.... Maybe this small and mysterious exchange of gifts remained inside me also, deep and indestructible, giving my poetry light.*
>
> —"The Gift," pages 281–282

Hooray for Literary Theory

The postmodern life is one of stories and personae. Literary theory—the study of how literature and art, in some cases, works—forms one set of techniques for understanding the parts of our lives in common with technology. One favorite concept is topos.

"Topos" is a term used in poetics and can be defined this way:

> *a conventionalized expression or passage in text which comes to be used as a resource for the composition of additional texts*
>
> —The New Princeton Encyclopedia of Poetry and Poetics

"Topos" literally means a place or location in Greek, and a topos is a place from which similar stories can be woven. An example from literature is the Garden of Eden. Anyone in a Western Judeo-Christian culture asked to tell a story about the Garden of Eden would likely come up with a story that was consistent with the "vision" of creation the Garden of Eden presents.

Topoi we're already familiar with include the shared purposes needed to build a cathedral or one of Dee Hock's chaords.

The Mob Effect Can Take Over, and Unexpected Plenitudes Ripen

The open-source community asked one question and answered it in a provocative way. They asked: "What if what once was scarce is now abundant?"

With their licenses they have found a gate through which some of us have passed. Christopher Alexander said one very important thing that is relevant to us today:

> *And yet the timeless way is not complete, and will not fully generate the quality without a name, until we leave the gate behind.*
>
> —"The Timeless Way of Building," page 529

It is only when we forget the ideas behind building something wonderful that we can actually do the building that makes things wonderful. Time and again I've heard poets and musicians talk about the letting go required to make true art or music. The battle to let go is the battle with the duende. It leads us to the place where order meets chaos, where the phase transition between stability and flexibility makes things happen.

Unfortunately, the open-source community is extremely conservative, focusing solely on the need to build up slowly a parallel open infrastructure

next to the proprietary ones already in place. They are working on an operating system; development tools like compilers, development environments, version control systems, and bug tracking systems; web servers; e-mail and other communications tools; a desktop; productivity tools; web browsers; and a small host of suchlike.

The image I have is of hackers encamped just outside a stone gate, carefully but joyfully building, well, cathedrals, just like those within the town. From a long distance, they appear huddled by the gate as if frightened by what may lie beyond in the unknown. It is a slightly bitter image.

Their philosophy is to build up a single reliable layer. Although the classic open-source licenses permit forking, it rarely happens because a fork is a failure—there is a right place to go and a right thing to build. The design center is whatever the hacker community likes, because the motivation is to create a world of open source that companies can never take away from them. It's a matter of liberation, and that's why it can be hard for companies to launch successful open-source projects.

One difference between open source and mob software is that open source topoi are technological while mob software topoi are people centered. The Jini topos speaks of a world of "simply connect" and "spontaneous networks" accompanied by a set of home, office, and automobile scenarios derived from this topos. The concepts of "simply connect" and "spontaneous networks" address how people relate to the technology. The Jini topos was highly effective and created a thriving community and associated technology—as hoped—along with thriving E-Speak, UPnP, and SOAP communities and associated technologies. Jini was one of the first explicitly mob-software projects.

The Apache topos speaks of web servers and technology for making them effective. Consequently, while Jini projects tend to fan out creating new services for people to use, Apache projects tend to focus on adding web server functionality and adopting related technologies such as Java Server Pages, servlets, and XML.

Open-source projects tend to be convergent: Each project is aimed at a particular artifact in the end. Mob-software projects tend to be divergent: The goal is to build variety and diversity so that competitive forces and natural selection in the user realm can take over.

Both open-source and proprietary software projects are internally highly ordered: In both cases there are only a few developers in core teams; in open source, there is a halo of occasional or opportunistic developers who chip in work, fix bugs, and make small changes. Proprietary projects are tightly managed by command-and-control systems; open-source projects have small core teams led by module owners who are strict gatekeepers. Both open-source and proprietary

commercial software projects populate the world in an orderly fashion. Proprietary software projects rationally pick and choose what products to make based on business goals, and hope scarcity doesn't kill them before they finish. Open-source projects pick and choose what to build based on technological needs and desires, and fear scarcity of interested people.

Rarely in either regime is the flash of duende followed by natural selection as strong as we'd hope for—as strong as in a living system. Mob software is about the boundary between order and chaos where life emerges and thrives. What is it like, the landscape of mob software?

~

Picture this: All devices that include computing elements are connected, and their collective software forms one large system. Pieces and chunks of this system sometimes go down or disconnect—later they might rejoin. The collective software works together even though its parts exist in multiple versions of different ages. Taken in the large, the software is continuously being upgraded and downgraded. Some parts of the software are performance only; other parts are private copies. Almost all the source code for this massive system—estimated in the billions of lines of code—is available under a license that grants total recombination rights: Any fragment of source code can be used for any purpose.

Companies dip into this pool to collect pieces and parts to build custom software for their own use. Doing this reduces their development costs to a fraction of what they once were. Following the precepts of xenia, these companies, for the most part, return their custom code to the pool. Other companies dip into this pool to create niche software; they augment their businesses by supporting this software and helping their clients.

The foibles of modular parts are minimized because each potential part is scrutinized and accumulates usages and annotations about its applicability and limitations. The part becomes productized, more like a library than a one of a kind; its size and scope is adjusted—in short, it evolves from many uses and public scrutiny. Unlike a stray bit of code which needs to be inspected and understood before it is reused, its constant and varied use provides confidence.

Customization is easy. Ordinary people trade and share customizations as easily as they trade and share recipes and thank-you-note templates. Customizations are largely data and schemata which are used to generate customized programs—just as databases today are so generated. Many companies do customizations for various segments of the population who pay modest amounts for it. Special interest groups perform customizations for their members—specialized versions of software are created for the disabled, the exceptional, different vocations, and for different work and play situations.

These customizations are not merely cosmetic, but go deep into the software. The variation in user interfaces for the same application is wider than the difference between punched cards and virtual reality. The end-user has been brought into the design process, creating a whole new range and variety of business opportunities. No more are interfaces designed for the hypothetical "average" person. Each person can have his or her own interface without imposing on an overworked development team.

Programming is mostly a process of adaptation partially automated by swarms of small bots that perform tasks such as interface evolution.

All software development takes place on the common corpus of source code by a mob of volunteers and developers paid by companies who reap benefits from the source base. Developers work primarily on projects of particular interest to them. Allowing people to choose their domains of expertise encourages artistry and craftsmanship. Mentoring circles and other forms of workshop are the mainstay of software development education. There are hundreds of millions of programmers.

The literature of code and its study spawns a passion for beauty and diversity in software which combined with a deep interest in user experience sponsors a creativity heretofore rarely observed in software development. Risk-taking and a willingness to open one's eyes to new possibilities and a rejection of worse-is-better make an environment where excellence is possible. Xenia invites the duende, which is battled daily because there is the possibility of failure in an aesthetic rather than merely a technical sense.

These developments arise from the organization of the community built around source code. This community has no fixed organization, nor is there a standard or typical way to work within it. Projects spring up, fork, spawn other projects, mutate, continue on, shut down; experimentation is encouraged; rarely are standards set, but when they are, the process is adapted to the situation while respecting the basic rights of community members. Like Dee Hock's chaords or the Jini Community, new parts of the organization are designed by its participants as needed, the design being informed by principles and pattern languages. Rights, principles, and ethics—for both individuals and corporations—are encoded in these community principles and pattern languages.

The community has a culture similar to the current open-source culture, but it is much more accepting of users, especially novice users—it's as if the spirit of xenia has raised to the highest status everyone with a stake in the community. Users participate in design at all levels of scale, and projects are begun specifically to address the needs or wants of a particular user community.

Moreover, experimentation and risk-taking emerge from the individual small acts of the community acting as a group mind. Because a basic infrastructure has

already been laid down, the freedom to pursue new ideas has become possible. Resource scarceness created by artificial boundaries no longer exists, and in an era of abundance, excess thrives.

Users and others create a new project by creating a topos for it—this is very hard work. By expressing a shared purpose, the topos, when honed and turned, becomes the attractor for developers to join in building its vision, in making it real. The topos as a story-making story attracts and provides room for experimentation and variation beyond what the initiators envisioned.

Forces That Get Us There

This sounds fantastic—I mean, it sounds like a fantasy. But a fantasy like this happened within the last 10 years, and another is under way.

The World Wide Web was first envisioned as a way to publish and cross-reference scientific papers. It was based on some of the least attractive technology ever to come down the road. If, somehow, it were possible to limit our view to web pages created by technologists and technology lovers—computer company sites, open-source sites, online computer 'zines—we would observe mediocrity, homogeneity, and bland. Let's face facts: As artists, we're good programmers.

But the technology was accessible to real people, and the Web grew almost boundlessly. Many sites are unbelievable: sites by artists, writers, galleries, glass blowers, urban diarists, monasteries, punks, the Vatican, architects, people with weird pets, pornographers, gnostics, stores, and children. We have cameras: Iguana Cam, Madagascar Hissing Cockroach Cam, Naked Molerat Cam, Corn Cam, and a range of cams best left imagined.

The lesson is simple: When opened up to real people, technology can blossom. The world envisioned by technologists is limited and flat, like the world of dog breeds when compared to the real diversity of species the world affords.

Under way is open source, demonstrating it is possible to do high-quality and effective, yet wildly distributed development with small snips of self-organization and freely available source code.

As companies join open-source projects over the next few years, communities increasingly will be built that include end-users of all stripes, and as desktop open-source systems are built, there will be increasing pressure brought to bear by the end-users for better usability, realistic customizability, and participation rights.

Performing software will put pressure on software development philosophy, and practitioners of performing software will more quickly see that we need to consider version maintenance and schema evolution applied to software modules.

As the code literature grows, we can start a literary discipline based on it and begin teaching code artistry once again—a literature of source code, user interfaces, and execution performance. When we do this we'll find that the durability, utility, and beauty of our software begins to take off until it isn't silly to consider it a literature.

Economic sea changes don't come that often—and we're in the middle of one right now. Art, publishing, and music businesses are all changing, as are our ideas on copyright and intellectual property. The economic exploitation of artists and writers is relatively new, and perhaps it won't survive the drift back to the gift economy we're witnessing. The names are silly, but for every Metallica against online sharing there's a Limp Bizkit in favor. Stephen King is directly publishing on the Web. Many of my poet and fiction writer friends are self-publishing or at least self-promoting and self-distributing using Amazon and all the rest. This change is happening in the software world too, with some major computer companies heavily into open source.

As in 6th century BC Greece, we may end up with a hybrid of the gift and commodity economies—find ways to adapt.

I've spoken of gifts, swarms, mobs, repair, biology, failures, and the duende. Mention of master planning, software process maturity levels, modular parts, mathematical reasoning, and all the other trappings of mainstream software development were either snide, sarcastic, demeaning, or insulting. Software skill is a gift from an unknown and unknowable source—perhaps the skill is an artistic talent, perhaps there are parts of the process of writing software that can be taught.

We can never move forward—we can never move forward—while the bulk of software work is done in secret and while wholly inadequate resources are given to it. Currently, almost all software development work is carried out in a way that cannot result in beautiful, living, adaptable artifacts. Unlike our friends in the built world, we just started to populate the landscape with beautiful software neighborhoods when the air hand of the commodity economy moved in like William Levitt's bulldozers and assembly-line house stampers.

The first Levittown—once called Island Trees—was a postwar innovation in low cost housing in which a potato patch on Long Island was turned into the first mass-produced housing tract. After the foundation was laid, specialized crews would descend on each lot one after another so fast that 18 houses were completed in the 8 to noon shift—one could say they used an especially brutal and efficient pattern language. Many lamented the approach. Peter Bacon Hales wrote of the criticism:

The accusations against Levittown from the first focused on its relentless homogeneity, the cramped quarters of its interiors, and the raw, unfinished quality of its landscape.

—http://www.uic.edu/~pbhales/Levittown/, Peter Bacon Hales

Architects, artists, and even some computer scientists have used Levittown as an example of how modernism—belief in reductionism and the ultimate value of the machine—can lead to the deathlike morphology of late 20th and early 21st century life, the unimaginably dulling effect of sameness and inhumanity on ordinary lives. I might have argued that, being made of nothing but modular parts laid out on curved roads, Levittown metaphorically epitomized what I find frightening and discouraging about contemporary software design and construction.

~

Postwar Americans needed affordable housing near jobs to raise the first wave of baby-boomers, not unaffordable aesthetics. The houses were snapped up while trucks hauling tools and pulling trailers carrying bulldozers drove away toward their next project. The architects, designers, builders, and developers did not care to learn from their projects, and the United States experienced an unending string of housing projects ever since that imposed a planned living experience on people bent more on living than on planning.

Levittown relied on wood framing, but many other projects cast their designs literally in concrete. The Pruitt-Igoe apartments in St. Louis won architectural awards in the mid-1950s for low-cost housing design. But the two complexes were simply steel and concrete highrise warrens in the mold of the Swiss architect Le Corbusier who said

a house is a machine to live in

—"Cities of Tomorrow," Peter Hall, page 205

and

the design of cities is too important to be left to the citizens

—"Cities of Tomorrow," Peter Hall, page 207

Seventeen years after being completed, the Pruitt-Igoe complex was dynamited, creating a vacant lot still mostly vacant, signaling the beginning of the postmodern era.

So in 1951, the designers of Levittown—designers just like us—watched as families poured in to begin life in the carefully planned and constructed warrens relentlessly devised according to modernist principles of machine love.

The world, however, did not wait, but soon observed what followed on. Nature, the Levittown community, and the duende did not care about the master plan: The bulldozed landscape began to sprout trees and shrubs, and over a period of years as economic realities changed, the community developed a sense of innovation, and its inhabitants a control of their own destinies. Not being trained as architects or builders did not faze them. They thought locally and acted locally. The community, through small acts of repair, transformed a homogenized and orderly Levittown into a place deserving the name Island Trees. Customization, additions, remodels, landscaping, disorder—the community, acting as a multitude or aggregation of persons regarded as not individually important—as a mob—made Levittown over.

Let me finish by again quoting Peter Bacon Hales, art historian from the University of Illinois at Chicago, who wrote the following for the mob-infested World Wide Web:

> *The raw, new quality of the landscape, too, didn't seem so awful to new renters and (a little later) owners, who knew that the trees and grass would quickly grow, and who understood the Levitt salesman's pitch promising opportunities to personalize the interior and exterior of your Levittown house. Life* [magazine] *ran a contest, seeking the best-decorated Levittown house, and the winner was a rather startling red-themed Mandarin-Revival Sino-Asian extravaganza. Over time, Levittown houses changed character, as their occupants rose in status and in economic wealth, and as families expanded and community standards of innovation and growth trickled from the home-improvement seminars at the Community Center and later the High School, out into the Saturday projects and summer vacation plans of Levittown residents. Today's heterogeneous Levittown is a testimony to the resilience of the community....*
>
> —http://www.uic.edu/~pbhales/Levittown/, Peter Bacon Hales

Where I'm from, the birds sing a pretty song and there's always music in the air.