Money Through Innovation Reconsidered

I would guess that fans of capitalism generally believe that innovation is the critical success factor and that natural evolution derived from competition makes the world better for consumers.

Every word of that belief is true, but the assumed meaning is not. Rather, the assumed meaning is that companies are rewarded for *their* innovation where innovation means invention, that the more innovation the better, and that consumers get a better world by leaps and bounds. Let's look at some evidence. The first is the editorial by Andrew Binstock quoted in "Into the Ground: C++":

> *The net effect shows that the forces of capitalism are much like those of evolution: the fittest survive. And when the fittest leap forward in their abilities, many weak or sick competitors are driven off or eaten. The benefits are clear: consumers such as you and me have better tools.* (Binstock 1994)

No less an authority than the editors of the *Economist* appear to share this belief. In a lead editorial on Bill Gates and Microsoft they wrote the following:

> *Antitrust policy must try to strike a balance between allowing market leaders to profit from their innovations and stopping them from abusing their power.* ("How Dangerous" 1995)

In a backing piece in the same issue they wrote:

> ***Innovation.*** *Computing, telecoms and the media have all seen rapid technological change. A stream of new ideas and products has sustained vigorous competition in most areas of these industries, driving costs and prices down. Thus, even if the product market were monopolised, trustbusters could afford to be sanguine if producers of new*

> *ideas were still able to make their way to market, or at least to force dominant companies to keep innovating and cutting prices themselves. But if such innovations also became monopolised, antitrust authorities would be right to worry.* ("Thoroughly Modern Monopoly" 1995)

The first problem is thinking that innovation means invention. Innovation means "to change a thing into something new, to renew, or to introduce something for the first time into the marketplace." The *Economist* seems to understand this partly because their lead editorial on Gates says this a few lines above the quoted passage:

> *As a young Harvard drop-out, he gladdened every nerd's heart by selling IBM a software system [MS-DOS] that his own tiny company, Microsoft, did not even own.* ("How Dangerous" 1995)

On the other hand, the backing piece refers to innovations as "a stream of new ideas."

But even in innovation there is a degree of improvement implied by "alter" or "make new," which is the root meaning of the word. I'll adopt the term *radical innovation* to mean an innovation based on new ideas rather than on incremental improvements to known ideas.

The second problem is in thinking that innovation causes the world to greatly improve for consumers, as witnessed by Binstock's comment, "when the fittest leap forward in their abilities."

My thesis is simple: Invention, radical innovation, and great leaps forward do not produce revenue commensurate with the required effort in the software industry, and invention usually doesn't work in any industry.

Before we get too general or abstract, let's look at some examples, starting with Microsoft. First, as the *Economist* points out, Microsoft didn't even own DOS when it sold it, so it did not invent it. Moreover, at the time IBM innovated—*introduced to the marketplace*—MS-DOS, its technology was at least 15 years old, perhaps older. Even if we look at Windows 95, which is just coming out as I write this, the newest ideas in it date to the mid-1970s, and the core of it is still rooted in inventions of the 1950s and 1960s. Windows 95 is an innovation in the true sense because it is a renewal of those old ideas in the current context and contains some improvements over them, yet the ideas are the same. It contains no stream of new ideas.

To be precise, the core operating system ideas of multitasking and threading, address space protection, and file system organization were well known by 1965. The idea of window-based user interfaces was well known by 1975, the ideas of hypertext (that's what OLE really is), mouse interaction, and menus were known in the late 1960s. Pop-up menus and drag-and-drop were invented in the early 1970s, I believe.

The artwork is new; the attention to detail in the interface is important; the upgrade to a modern computing platform makes the ideas effective rather than merely amusing; and the documentation and on-line help systems make it all palatable.

In short, there are plenty of reasons that Windows 95 is good for consumers—it moves the technology center of gravity up to the late 1960s from the late 1950s—but few of those reasons have to do with invention or radical innovation.

I believe Microsoft is pursuing the correct strategy for a technology company, but this strategy has nothing to do with invention, radical innovation, creativity, or imagination.

Let's look at UNIX. UNIX is an operating system written as a free alternative to then-existing operating systems with multitasking capabilities, such as Multics, in contrast to whose name the name UNIX derived. UNIX was developed at Bell Labs and later major improvements were made at the University of California at Berkeley. A copy of UNIX went out on basically every tape that was delivered with a PDP-11 and later with every VAX as freeware. Universities used it because it was free and it was good enough and because the sources were included so that local budding wizards could customize it to local use.

From the mid-1970s until nearly the mid-1980s, UNIX was unknown outside the university setting. But around 1981 a group of grad students, graduates, and faculty from Stanford and Berkeley and some others founded Sun Microsystems (SUN was an acronym for *S*tanford *U*niversity *N*etwork, which was the project for which the workstation later known as the Sun 1 was built) and chose to use "standard" or, at any rate, existing components: the Motorola 68010 processor, Berkeley UNIX, and the Stanford networking hardware.

Now UNIX is the default workstation operating system.

When UNIX first came out there was nothing inventive about it; in fact, in some ways, it was poorly engineered. A good example of this is the PC-losering problem we encountered in the essay "The End of History and the Last Programming Language."

The PC-losering problem occurs when a user program invokes a system routine to perform a lengthy operation that might require significant state, such as IO buffers. If an interrupt occurs during the operation, the state of the user program must be saved. Because the invocation of the system routine is usually a single instruction, the program counter (PC) of the user program and the registers do not adequately capture the state of the process. The system routine thus must either back out or press forward. The right thing is to back out and restore the user program PC to the instruction that invoked the system routine so that resumption of the user program after the interrupt, for example, will reenter the system routine. It is called *PC-losering* because the PC is being coerced into "loser mode," where "loser" was the affectionate name for "user'" at MIT when the problem was solved there in ITS, one of the MIT operating systems of the 1960s and 1970s.

ITS has several dozens of pages of assembly language code dedicated to backing out of a system call so that the normal context-saving mechanism of saving the PC and registers will work. The trade-off was made that implementation simplicity was sacrificed so that the interface to the system in the presence of interrupts would be preserved: Programmers need know only that when an interrupt occurs and has been handled, the main computation can be resumed by restoring the registers and jumping to the saved PC.

In UNIX as of the late 1980s the solution is quite different. If an interrupt occurs during a system call, the call will immediately complete and will return an error code. The trade-off was made that the simplicity of the interface to the system would be sacrificed so that implementation simplicity would be preserved, and therefore the implementation could be more easily checked for correctness: Programmers must insert checks for error codes after each system call to see whether the system call actually worked.

It might seem odd that one would write a system call and then have to explicitly check whether it worked—it returned, didn't it? But, UNIX doesn't need that 60 or so pages of complex and probably incorrect assembly language code that backs out of every system call, and therefore it was easier to make UNIX portable—UNIX was simpler and smaller.

UNIX chose implementation simplicity over interface simplicity, whereas while ITS chose interface simplicity. As you might suspect, many UNIX programs were incorrect because after an interrupt they would crash, but UNIX is an operating system still in existence and ITS is not.

Over the years, with so many people adopting UNIX, it was improved so that now it is a fairly good operating system—consider Mach and Solaris. The pattern is to start modest and improve according to the dictates of the users.

There are many examples of this pattern in industry. Japanese automobile manufacturers started by producing low-cost, moderate-quality automobiles for sale in the United States, and as they were accepted the manufacturers upgraded their product lines with improvements. It was well known in the 1970s how to make excellent-quality cars with lots of features—all you had to do was look at European cars—but that approach would have resulted in minimal acceptance. No one wanted to buy a luxury car or even a regular car from a company named Toyota or Nissan. But they might want to buy a minimally equipped, moderate quality car at a low price.

The pattern is to get something of acceptable quality into the largest marketplace you can and then later improve or add a narrower focus with higher-quality, more inventive products. Today many people look first to Japanese automobile manufacturers for luxury cars as well as for cars at all levels of quality and features.

In 1990 I proposed a theory, called *Worse Is Better*, of why software would be more likely to succeed if it was developed with minimal invention (Gabriel 1990). Here are the characteristics of a worse-is-better software design for a new system, listed in order of importance:

- *Simplicity*: The design is simple in implementation. The interface should be simple, but anything adequate will do. Implementation simplicity is the most important consideration in a design.
- *Completeness:* The design covers only necessary situations. Completeness can be sacrificed in favor of any other quality. In fact, completeness must be sacrificed whenever implementation simplicity is jeopardized.
- *Correctness:* The design is correct in all observable aspects.
- *Consistency:* The design is consistent as far as it goes. Consistency is less of a problem because you always choose the smallest scope for the first implementation.

Implementation characteristics are foremost:

- The implementation should be fast.
- It should be small.
- It should interoperate with the programs and tools that the expected users are already using.
- It should be bug-free, and if that requires implementing fewer features, do it.
- It should use parsimonious abstractions as long as they don't get in the way. Abstraction gets in the way when it makes implementation too hard, too slow, or hides information that shouldn't be hidden. (I once heard an interesting comment—sort of a motto—at a scientific computing conference: Abstractions = page-faults.)

It is far better to have an underfeatured product that is rock solid, fast, and small than one that covers what an expert would consider the complete requirements.

These are the benefits of a software system designed and implemented this way:

- It takes less development time, so it is out early and can be adopted as the de facto standard in a new market area.
- It is implementationally and functionally simple, so it can run on the smallest computers. Maybe it can be easily ported as well—if it uses a simple portability model. At any given time the mainstream computer users—whether individuals or corporations—are running hardware at least two generations old.

- If it has some value, it will be ported or adopted and will tend to spread like a virus.
- If it has value and becomes popular, there will be pressure to improve it, and over time it will acquire the quality and feature-richness of systems designed another way, but with the added advantage that the features will be those the customers or users want, not those that the developers think they should want.

The path of acceptance is that the worse-is-better system is placed in a position in which it can act like a virus, spreading from one user to another, by providing a tangible value to the user with minimal acceptance cost. There is little question it can run everywhere, so there are no artificial barriers. The system can be improved by the end users because it is open or openly modifiable, and its implementation is simple and habitable enough that users can do this.

This simple implementation provides a means for the originators to improve the system over time in a way consistent with how it's being used, thereby increasing the magnitude of the most important evaluation factors. Over time, the system will become what lucky developers using a more traditional methodology would achieve.

This may seem counterintuitive—many people believe that being competitive requires doing the absolute best development you can. The following is a characterization of the contrasting design philosophy, which I call *The Right Thing*:

- *Simplicity:* The design is simple, both in implementation and interface. Simplicity of implementation is irrelevant.
- *Completeness:* The design covers as many important situations as possible. All reasonably expected cases must be covered.
- *Correctness:* The design is correct in all observable aspects. Incorrectness is simply not allowed.
- *Consistency:* The design is thoroughly consistent. A design is allowed to be slightly less simple and less complete in order to avoid inconsistency. Consistency is as important as correctness.

There really is only one implementation requirement:

- It should use hard abstraction throughout. Elements must be properly encapsulated in order to achieve information hiding.

The acceptance path for a right-thing system is that it is the right thing off the bat, and even though it came late to the marketplace, it is so wonderful that it is quickly accepted. Of course, it has to be on every necessary platform either right away or quickly. This scenario can happen, it is simply unlikely.

One of the more familiar examples of a right-thing system is the Macintosh and the Macintosh operating system. When the Mac was introduced in 1984 it was arguably the product of largely right-thing design though worse-is-better implementation. It used ideas and technology only 10 years old or so, and it was considerably ahead of its competition. One might say that Apple tried to run up the score on the competition by producing a product so far in advance as to be almost not in the same competitive milieu.

Although one could argue that the Macintosh today—some 10 years later—is a success, it enjoys only about 10% to 15% of the total PC market. It is not used in the mainstream; rather, its strongholds are primarily in desktop publishing, graphic design, and marketing organizations, whereas Microsoft platforms are used throughout the corporation. There are many good reasons for this failure to capture more of the market in addition to the problems of the right-thing philosophy, which we'll look at further. MS-DOS was licensed to all the PC clone manufacturers, so that consumers had a choice of a variety of price and performance characteristics. Of course, IBM, having strong influence among MIS departments, helped seed the PC in mainstream settings.

The popularity of the PC and DOS created the opportunity for a market of software on this platform: A company wishing to write and sell a software product always looked to the PC and DOS first because it had a larger market share to start with because IBM shipped DOS, other manufacturers did too, and they all shipped the same platform.

The difference between what happened to DOS and the Macintosh is an example of what is known as *gambler's ruin*. Suppose two players have equally large piles of pennies—pick a round number like a jillion. If the players have a fair coin and flip it, with one particular player taking a penny from the other if heads, vice versa if tails, what will happen? It would seem that the players would keep their positions of roughly equal amounts of pennies. But in mathematical terms, this is an event of measure 0—it will never happen. The probability is essentially 1 that one player will lose all his or her pennies. But if one of the players starts out with more pennies than the other, the probability that that player will end up with all of them is equal to the proportion of that player's pennies to the total number of pennies.

There is a tendency to think that somehow the free market is fair and that all seemingly fairly matched competitors will survive. If there is some aspect of a business that involves consumable resources used in competition, that business is subject to gambler's ruin.

In the case of PCs versus Macintoshs, the PC/DOS player started with many more pennies and won. However, there is more to it than chance. The Macintosh was clearly superior, yet it lost. Among the reasons for losing was that Apple didn't license its technology until 1995 so that a clone market didn't appear earlier (and one could say that waiting 11 years to do it was a suboptimal strategy).

There are other problems stemming from the difference between the two philosophies. One is that MS-DOS provided a simpler, more malleable base so that the aftermarket could more easily mold MS-DOS into what it needed. Once those characteristics of improvement were known, MS-DOS evolved.

❧ ❧ ❧

One of the obvious criticisms of worse-is-better is that there is something worse going on, that is, something not good or not right. It seems like the idea is intentionally to put out an inferior product and then hope that things go well. This isn't the case, but even this argument itself can be attacked.

When you say that a system is inferior to what it could be, you are basing that conclusion on an evaluation criterion. Necessarily you are using, perhaps intuitively, a metric based on a projection. That is, no one can evaluate an entire system in all of its use settings, so you are looking at either only part of the system or some of the settings, that is, a projection.

The problems with this are that a projection always loses information and the choice of projection is typically based on bias. For example, Modula-3 is a great programming language that is unsuccessful. Proponents of Modula-3 believe that it is a fabulous language and better, in particular, than C++. Modula-3 is based on a Cfront-like compiler, runs on a lot of computers, and is free. These are characteristics it shares with C++. Unlike C++, Modula-3 is optimized for mathematical and abstractional cleanliness. Moreover, it doesn't have a very good compiler, it lacks an environment, and it has no compelling systems written in it that people want to use and could use in such a way to see and appreciate the advantages of Modula-3. This last point can be understood by thinking about how much more likely it would be for people to use Basic English or Esperanto if there were a compelling piece of literature written in it.

Most important, the evaluation function—mathematical cleanliness—is not valued by the users, and perhaps they can't understand why it would be valuable except in theoretical terms.

The PC-losering example can be viewed this way, too. The characteristic valued by the ITS designer is simplicity of interface, which has led, in this case, to a large, complex system that can run on only a specific computer (the PDP-10) because it is written in assembly language *because* that is the only way to write it so that it's fast enough. UNIX is written in C, which is portable, and UNIX was simple enough early on that even this implementation choice offered acceptable performance.

In the modern era, we have come to favor simplicity over complexity, perfection over imperfection, symmetry over asymmetry, planning over piecemeal growth, and design awards over habitability. Yet if we look at the art we love and the music, the buildings, towns, and houses, the ones we like have the quality without a name, not the deathlike morphology of clean design.

In many cases, the software system that succeeds starts with a kernel of good technology, but it is not committed to fully to realizing that technology. By enabling users to live in the implementation to some extent or to mold the technology to users' needs while dropping hurdles to the use of the technology, that system is able to survive and evolve.

The right-thing philosophy is based on letting the experts do their expert thing all the way to the end before users get their hands on it. The thought is that until the whole grand structure is built and perfected, it is unusable, that the technology requires an expansive demonstration to prove its effectiveness, and that people are convinced by understanding the effectiveness of a technology, as if we needed to know that vehicles based on internal combustion engines could carry immense weights—tons and tons—or seat dozens of people before someone would be willing to buy a four-person, 15-horsepower car.

Worse-is-better takes advantage of the natural advantages of incremental development. Incremental improvement satisfies some human needs. When something is an incremental change over something else already learned, learning the new thing is easier and therefore more likely to be adopted than is something with a lot of changes. To some it might seem that there is value to users in adding lots of features, but there is, in fact, more value in adding a simple, small piece of technology with evolvable value.

The goal of a software enterprise is to make it into the mainstream, and being in the mainstream does not mean selling to a particular set of corporations. It means selling to customers with particular characteristics.

One of the key characteristics of the mainstream customer is conservatism. Such a customer does not want to take risks; he (let's say) doesn't want to debug your product; he doesn't want to hit a home run so he can move up the corporate ladder. Rather, he wants known, predictable improvement over what he is doing today with his own practices and products. He wants to talk to other folks like himself and hear a good story. He doesn't want to hear how someone bet it all and won; he wants to hear how someone bought the product expecting 10% improvement and got 11%. This customer is not interested in standing out from the crowd, particularly because of a terrible error in his organization based on a bad buying decision. These aren't the characteristics of someone who would love technology and build an extravagant product based on it; therefore, it is hard for a technologist to understand what this customer values.

The ideal of the free market supports this kind of growth. If you decide to spend a lot of resources developing a radical innovation product, you may be throwing away development money. Why bet millions of dollars all at once on something that could flop when you can spend a fraction, test the ideas, improve the ideas based on customer feedback, and spend the remainder of money on the winning evolution of the technology? If you win, you will win more, and, if you lose, you will lose less. Moreover, you will be out there ahead of competition which is happily making the right-thing mistake.

Microsoft won by putting out junk with a core good idea—and usually not their own idea—and then improved and improved, never putting out something too innovative, too hard to swallow, too risky.

When you put out small incremental releases, you can do it more frequently than you can with large releases, and you can charge money for each of those releases. With careful planning you can charge more for a set of improvements released incrementally than the market would have borne had you released them all at once, taking a risk on their acceptance to boot. Moreover, when you release many small improvements, you have less risk of having a recall, and managing the release process also is easier and cheaper. With incremental improvement, the lifetime of an idea can be stretched out, and so you don't have to keep coming up with new ideas. Besides, who wants to base one's economic success on the ability to come up with new ideas all the time?

The free market means improvement for the consumers, but at the slowest possible rate, and companies that try to go faster than that rate are almost always hammered down or killed. On top of these factors is the feedback loop in the free market. We've seen some reasons that it makes business sense to make small improvements rather than large ones. Because companies use these techniques, the pattern is established that the free market goes slowly. Given that, consumers now act in accordance with the free market. For example, because consumers rarely see radical innovation—it's too expensive, so companies don't do it, and a series of incremental improvements seldom if ever amounts to a radical innovation—they suspects its value. You might say that consumers are conditioned by the free market against radical innovation.

When you add the risk aversion of the mainstream marketplace to the minimal innovation fostered by the free market, it is easy to see that the right-thing is just too risky to try unless you really are sure it will work.

The key idea is to identify the true value of what you bring to marketplace, to test those ideas in a small context, to improve them in a participatory manner, and then to package that value in the least risky way.

Let's look at the Lisp example. In order for Lisp to succeed as Lucid promoted it, the customer would have had to buy into AI; buy into using an unusual language; buy into the peculiar development environment with its own code construction, debugging, and maintenance tools; and accept the view that the standard computing world was at the other end of a narrow foreign function interface. This is too much to ask someone to accept when the value to the customer is a simple rule-based computing paradigm from AI and incremental development from Lisp. There is no reason not to have presented expert-system technology in more familiar surroundings or to introduce incremental development using a Lisp constructed more like a traditional programming language.

If you decide to go whole hog and produce a right-thing product, you may have made too much of an engineering commitment, certainly in terms of effort put into the product. If the product is not right, it will be too large and complex to modify to be more in line with customers' needs. So the very investment you might have thought was the way to put distance between you and possible competition by making the effort for entry too great is the very impediment to modifying your product to be what customers need.

When you release a modest product first, however, the sheer amount of typing necessary to adapt is small, and therefore can be done quickly, which means that your company is making rapid transitions in the marketplace, transitions your customers may require.

The ideal situation is that your proprietary value is small compared with the total size of your product, which perhaps can be constructed from standard parts by either you or your customers or partners.

There is a saying adapted from something Wade Hennesey said about his own way to making money:

Wade's Maxim: *No one ever made money by typing.*

What this means in our context is that you cannot try to build too large and complex product by yourself—you need to get parts from other places if your product is complex, and you must make your value clear and apparent, and it must be a simple decision for a customer to use your technology.

If you go out with a product without cachet, you might not like the results. *Cachet*—a word whose meaning has changed recently to include the characteristic of having a positive or desirable aura—is what will attract your customers to your product. For example, Netscape has cachet because it is the hip window into the Web.

Cachet is that which makes a system or language attractive over and separate from its objective features and benefits—it's why kids ask for particular toys for Christmas. There is nothing wrong with your customers acting like kids and bugging their parents—their managers—into buying them your toy.

Cachet is how you appeal to your early adopters, whose successes pave the way for acceptance into the mainstream. For many early adopters, technological innovation is enough, but you should make sure that the cachet is technological only and not based on overwhelming mass of technology. I use the term *acceptance group* to refer to people who must accept your product and technology in order for you to be successful—at first they are the early adopters, later the mainstream.

The larger you want your acceptance group to be, the lower technical skill level you have to aim for. This is not a matter of elitism, it's a matter of mathematics. And if you aim for a lower technical talent group, you have to provide a simple habitat—your system—for them to have to learn and inhabit.

Furthermore, the larger the acceptance group, the smaller and lower-power the computer they will be using. The less technologically advanced the target acceptance group, the less likely they will be to upgrade their computers unless there is a specific reason to do so. And it is unlikely your system will be the ultimate reason.

Nevertheless, any acceptance group needs to see that your system has the right mythos, the right cachet. Their first impression must be good in order to generate word of mouth, so it must have visible value right out of the box. If your system has a lot of complex functionality, it will be difficult to make sure the first look turns into a good first impression. The less functionality it has, the more controlled an initial look the users get. If you provide a free or cheap implementation with sources, then you might add a "wizard" component to your acceptance group, adding not only a free development arm but also cachet.

Don't be afraid to find the rhinoceros to pick fleas from. There is nothing wrong with attaching your fortunes to those of a platform manufacturer or other system provider whose star is bright and gaining brightness. For example, it has made sense for the last 10 years to bet your company on Microsoft by writing software for DOS and Windows. Especially smart is being able to use parts of the rhino's products and technology to fill out your product. The less innovation you do, the better. Run just fast enough to win, but don't ruin your chances in the next race.

Recall that Andrew Binstock invoked evolution:

The net effect shows that the forces of capitalism are much like those of evolution: the fittest survive. (Binstock 1994)

The popular idea even today is that evolution operates by making improvements—some large and some small. Darwin himself recognized that the effect of natural selection in making improvements was local at best. Darwin was writing his work when Britain was in the midst of its industrial expansion and its interest in showing that the free market led to improvements. Darwin was also under pressure to show that the press for perfection was inexorable from slime to people vectored through apes.

The theme of this theory is wedging out another species through competition—and this is just what Binstock believes is happening.

To respond to the pressure he was facing, Darwin came up with his wedge theory, which stated that the panorama of life was chock full of species occupying every ecological niche. Each species was like a wedge, and the only way that a new species could find a place in the world was to wedge out another one, and the way that happened was by being better somehow.

Undoubtedly this happens in many cases, but usually only in cases of small local improvements. For example, the human brain has increased in size since the first appearance of *Homo sapiens*, and this happened because a large brain had advantages, and so population dynamics selected for this trait—perhaps early men and women preferred to have intercourse with large-headed members of the opposite sex.

But this cannot explain how a large complex brain came into existence in the first place. Until something like a brain, only smaller, reaches a certain size and complexity, it has no purpose and doesn't really do anything. Therefore no selections are based on it except ones irrelevant to its ultimate use. This means that there was no reason not to grow increasingly larger brains—maybe such a brain simply belonged to someone more attractive. Eventually some environmental change occurred in which the large brain just waiting for something to do had a new purpose—language, for example—and its usefulness started to dominate natural selection.

To get to the point of understanding why brains can't just have grown and grown we need to look at mass extinctions. The best-known extinction was that of the dinosaurs. Dinosaurs and mammals overlapped by 100 million years, plenty of time for any wedge-related competition to have increased the mammals' niche or even forced out the dinosaurs if the mammals were so much better than dinosaurs. However, it took a disaster—currently believed to be a comet or asteroid hitting the earth—to dislodge the dinosaurs. The small mammals were able to survive the cloud-induced winter or the flash-induced heat that resulted but the dinosaurs were not. The mammals had something that enabled them to survive, and almost certainly it was something marginal or irrelevant until the catastrophic event—

otherwise you have to believe that evolution operates by the future influencing the past.

Natural selection is a mechanism to reject changes not to establish them. In fact, natural selection is used to slow down changes in the areas that matter—if a particular wing structure works, you don't want to try changing it. Therefore, those parts of an organism most central to its survival mutate over time the most slowly. In fact, biologists use the rate of mutation to determine whether some puzzling organ or structure has a purpose: The higher the rate, the less important purpose it has.

Only things that are relatively worthless change rapidly and dramatically. These worthless things provide variety, and someday an environmental change can make one of those changes or a related group of them important to survival. Then that one or that group will join the ranks of the natural-selection protected, and perhaps some formerly static parts may no longer be essential and can begin to change.

For example, we use tires for automobiles, but in Africa and India where cars aren't so numerous, people use them to make sandals. If a disaster occurred in which cars and industry disappeared, car tires might thus be used to shoe people. Nothing in the design or intended purpose of tires had anything directly to do with sandals, but quirks of the new environment make them useful for the new purpose. Then perhaps tires could begin to be improved for their new use.

Mass extinctions teach us this when we look at the details, but those details are too extensive to go into in this essay. I recommend *Eight Little Piggies* by Stephen Jay Gould (1993) for those interested.

In other words, natural selection provides very little incremental improvement. Rather, almost all dramatic change occurs because the environment changes and makes the worthless invaluable.

I think that this works in the marketplace as well. The simulation-based mechanisms of Smalltalk still are useful, but Smalltalk's success in the mid-1990s as a business-oriented language stems from the use of those mechanisms to model naturally, not simulate, simple business objects. This was not one of Smalltalk's design goals, yet it has turned out to be the key success factor.

Population dynamics also makes a difference. Dinosaurs coexisted with mammals for 100 million years, with the rat-sized mammals making no headway. And nothing about the mammals forced out the dinosaurs through competition.

The lesson is to put out your small technology nugget, not typing too much, and wait to see whether the environment changes so it has an interesting new use for your nugget and then change your product to take advantage of it and how it's used. Make a small, simple wing, wait 'til flying takes off, then build a flying machine.

❧ ❧ ❧

These ideas on innovation might seem odd, but they echo those of prominent economic thinkers. Roland Schmitt and Ralph Gomory wrote:

> *[The Japanese company] gets the product out fast, finds out what is wrong with it, and rapidly adjusts; this differs from the U.S. method of having a long development cycle aimed at a carefully researched market that may, in fact, not be there.* (Schmitt and Gomory 1988, 1989)

And L. M. Branscombe wrote this:

> *After carefully assessing the functional requirements and the permissive cost of a new product for consumer markets, the [Japanese] engineering team will work on a series of designs, incrementally approaching the cost target, testing for consumer acceptance. The initial design will very likely incorporate technical processes of a most sophisticated nature, taken from the best publications in science. But they will be used in an exceedingly conservative application. . . . When the product is introduced it already incorporates leading edge technologies with a high degree of extendability, but with little of the risk associated with pushing the limits of the technology too soon. American engineers faced with the same problem would spend much more time and money pushing the scientific level of the technology to a design point where it is clearly superior to proven alternatives before introducing it. . . . In all likelihood, their costs will be higher and risk of program slippage greater than the Japanese approach. And the technology in the product, when introduced, will be more difficult to extend incrementally.* (Branscombe 1987)

Remember: No one ever made money by typing.