Table of Contents

Programming Bottom-Up

1993

(This essay is from the introduction to On Lisp*.)*

It's a long-standing principle of programming style that the functional elements of a program should not be too large. If some component of a program grows beyond the stage where it's readily comprehensible, it becomes a mass of complexity which conceals errors as easily as a big city conceals fugitives. Such software will be hard to read, hard to test, and hard to debug.

In accordance with this principle, a large program must be divided into pieces, and the larger the program, the more it must be divided. How do you divide a program? The traditional approach is called *top-down design:* you say "the purpose of the program is to do these seven things, so I divide it into seven major subroutines. The first subroutine has to do these four things, so it in turn will have four of its own subroutines," and so on. This process continues until the whole program has the right level of granularity-- each part large enough to do something substantial, but small enough to be understood as a single unit.

Experienced Lisp programmers divide up their programs differently. As well as top-down design, they follow a principle which could be called *bottom-up design*-- changing the language to suit the problem. In Lisp, you don't just write your program down toward the language, you also build the language up toward your program. As you're writing a program you may think "I wish Lisp had such-and-such an operator." So you go and write it. Afterward you realize that using the new operator would simplify the design of another part of the program, and so on. Language and program evolve together. Like the border between two warring states, the boundary between language and program is drawn and redrawn, until eventually it comes to rest along the mountains and rivers, the natural frontiers of your problem. In the end your program will look as if the language had been designed for it. And when language and program fit one another well, you end up with code which is clear, small, and efficient.

It's worth emphasizing that bottom-up design doesn't mean just writing the same program in a different order. When you work bottom-up, you usually end up with a different program. Instead of a single, monolithic program, you will get a larger language with more abstract operators, and a smaller program written in it. Instead of a lintel, you'll get an arch.

In typical code, once you abstract out the parts which are merely bookkeeping, what's left is much shorter; the higher you build up the language, the less distance you will have to travel from the top down to it. This brings several advantages:

1. By making the language do more of the work, bottom-up design yields programs which are smaller and more agile. A shorter program doesn't have to be divided into so many components, and fewer components means programs which are easier to read or modify. Fewer components also means fewer connections between components, and thus less chance for errors there. As industrial designers strive to reduce the number of moving parts in a machine, experienced Lisp programmers use bottom-up design to reduce the size and complexity of their programs.

2. Bottom-up design promotes code re-use. When you write two or more programs, many of the utilities you wrote for the first program will also be useful in the succeeding ones. Once you've acquired a large substrate of utilities, writing a new program can take only a fraction of the effort it would require if you had to start with raw Lisp.

3. Bottom-up design makes programs easier to read. An instance of this type of abstraction asks the reader to understand a general-purpose operator; an instance of functional abstraction asks the reader to understand a special-purpose subroutine. [1]

4. Because it causes you always to be on the lookout for patterns in your code, working bottom-up helps to clarify your ideas about the design of your program. If two distant components of a program are similar in form, you'll be led to notice the similarity and perhaps to redesign the program in a simpler way.

Bottom-up design is possible to a certain degree in languages other than Lisp. Whenever you see library functions, bottom-up design is happening. However, Lisp gives you much broader powers in this department, and augmenting the language plays a proportionately larger role in Lisp style-- so much so that Lisp is not just a different language, but a whole different way of programming.

It's true that this style of development is better suited to programs which can be written by small groups. However, at the same time, it extends the limits of what can be done by a small group. In *The Mythical Man-Month*, Frederick Brooks proposed that the productivity of a group of programmers does not grow linearly with its size. As the size of the group increases, the productivity of individual programmers goes down. The experience of Lisp programming suggests a more cheerful way to phrase this law: as the size of the group decreases, the productivity of individual programmers goes up. A small group wins, relatively speaking, simply because it's smaller. When a small group also takes advantage of the techniques that Lisp makes possible, it can win outright.

New: Download On Lisp for Free.

[1] "But no one can read the program without understanding all your new utilities."

To see why such statements are usually mistaken, see Section 4.8.

Lisp for Web-Based Applications

After a link to [Beating the Averages](#) was posted on slashdot, some readers wanted to hear in more detail about the specific technical advantages we got from using Lisp in Viaweb. For those who are interested, here are some excerpts from a talk I gave in April 2001 at BBN Labs in Cambridge, MA.

▪

[BBN Talk Excerpts (ASCII)](#)

Beating the Averages

April 2001, rev. April 2003

(This article is derived from a talk given at the 2001 Franz Developer Symposium.)

In the summer of 1995, my friend Robert Morris and I started a startup called [Viaweb](). Our plan was to write software that would let end users build online stores. What was novel about this software, at the time, was that it ran on our server, using ordinary Web pages as the interface.

A lot of people could have been having this idea at the same time, of course, but as far as I know, Viaweb was the first Web-based application. It seemed such a novel idea to us that we named the company after it: Viaweb, because our software worked via the Web, instead of running on your desktop computer.

Another unusual thing about this software was that it was written primarily in a programming language called Lisp. It was one of the first big end-user applications to be written in Lisp, which up till then had been used mostly in universities and research labs. [1]

The Secret Weapon

Eric Raymond has written an essay called "How to Become a Hacker," and in it, among other things, he tells would-be hackers what languages they should learn. He suggests starting with Python and Java, because they are easy to learn. The serious hacker will also want to learn C, in order to hack Unix, and Perl for system administration and cgi scripts. Finally, the truly serious hacker should consider learning Lisp:

> Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

This is the same argument you tend to hear for learning Latin. It won't get you a job, except perhaps as a classics professor, but it will improve your mind, and make you a better writer in languages you do want to use, like English.

But wait a minute. This metaphor doesn't stretch that far. The reason Latin won't get you a job is that no one speaks it. If you write in Latin, no one can understand you. But Lisp is a computer language, and computers speak whatever language you, the programmer, tell them to.

So if Lisp makes you a better programmer, like he says, why wouldn't you want to use it? If a painter were offered a brush that would make him a better painter, it seems to me that he would want to use it in all his paintings, wouldn't he? I'm not trying to make fun of Eric Raymond here. On the whole, his advice is good. What he says about Lisp is pretty much the conventional wisdom. But there is a contradiction in the conventional wisdom: Lisp will make you a better programmer, and yet you won't use it.

Why not? Programming languages are just tools, after all. If Lisp really does yield better programs, you should use it. And if it doesn't, then who needs it?

This is not just a theoretical question. Software is a very competitive business, prone to natural monopolies. A company that gets software written faster and better will, all other things being equal, put its competitors out of business. And when you're starting a startup, you feel this very keenly. Startups tend to be an all or nothing proposition. You either get rich, or you get nothing. In a startup, if you bet on the wrong technology, your competitors will crush you.

Robert and I both knew Lisp well, and we couldn't see any reason not to trust our instincts and go with Lisp. We knew that everyone else was writing their software in C++ or Perl. But we also knew that that didn't mean anything. If you chose technology that way, you'd be running Windows. When you choose technology, you have to ignore what other people are doing, and consider only what will work the best.

This is especially true in a startup. In a big company, you can do what all the other big companies are doing. But a startup can't do what all the other startups do. I don't think a lot of people realize this, even in startups.

The average big company grows at about ten percent a year. So if you're running a big company and you do everything the way the average big company does it, you can expect to do as well as the average big company-- that is, to grow about ten percent a year.

The same thing will happen if you're running a startup, of course. If you do everything the way the average startup does it, you should expect average performance. The problem here is, average performance means that you'll go out of business. The survival rate for startups is way less than fifty percent. So if you're running a startup, you had better be doing something odd. If not, you're in trouble.

Back in 1995, we knew something that I don't think our competitors understood, and few understand even now: when you're writing software that only has to run on your own servers, you can use any language you want. When you're writing desktop software, there's a strong bias toward writing applications in the same language as the operating system. Ten years ago, writing applications meant writing applications in C. But with Web-based software, especially when you have

the source code of both the language and the operating system, you can use whatever language you want.

This new freedom is a double-edged sword, however. Now that you can use any language, you have to think about which one to use. Companies that try to pretend nothing has changed risk finding that their competitors do not.

If you can use any language, which do you use? We chose Lisp. For one thing, it was obvious that rapid development would be important in this market. We were all starting from scratch, so a company that could get new features done before its competitors would have a big advantage. We knew Lisp was a really good language for writing software quickly, and server-based applications magnify the effect of rapid development, because you can release software the minute it's done.

If other companies didn't want to use Lisp, so much the better. It might give us a technological edge, and we needed all the help we could get. When we started Viaweb, we had no experience in business. We didn't know anything about marketing, or hiring people, or raising money, or getting customers. Neither of us had ever even had what you would call a real job. The only thing we were good at was writing software. We hoped that would save us. Any advantage we could get in the software department, we would take.

So you could say that using Lisp was an experiment. Our hypothesis was that if we wrote our software in Lisp, we'd be able to get features done faster than our competitors, and also to do things in our software that they couldn't do. And because Lisp was so high-level, we wouldn't need a big development team, so our costs would be lower. If this were so, we could offer a better product for less money, and still make a profit. We would end up getting all the users, and our competitors would get none, and eventually go out of business. That was what we hoped would happen, anyway.

What were the results of this experiment? Somewhat surprisingly, it worked. We eventually had many competitors, on the order of twenty to thirty of them, but none of their software could compete with ours. We had a wysiwyg online store builder that ran on the server and yet felt like a desktop application. Our competitors had cgi scripts. And we were always far ahead of them in features. Sometimes, in desperation, competitors would try to introduce features that we didn't have. But with Lisp our development cycle was so fast that we could sometimes duplicate a new feature within a day or two of a competitor announcing it in a press release. By the time journalists covering the press release got round to calling us, we would have the new feature too.

It must have seemed to our competitors that we had some kind of secret weapon-- that we were decoding their Enigma traffic or something. In fact we did have a secret weapon, but it was simpler than they realized. No one was leaking news of their features to us. We were just able to develop software faster than anyone thought possible.

When I was about nine I happened to get hold of a copy of *The Day of the Jackal,* by Frederick Forsyth. The main character is an assassin who is hired to kill the president of France. The assassin has to get past the police to get up to an apartment that overlooks the president's route. He walks right by them, dressed up as an old man on crutches, and they never suspect him.

Our secret weapon was similar. We wrote our software in a weird AI language, with a bizarre syntax full of parentheses. For years it had annoyed me to hear Lisp described that way. But now it worked to our advantage. In business, there is nothing more valuable than a technical advantage your competitors don't understand. In business, as in war, surprise is worth as much as force.

And so, I'm a little embarrassed to say, I never said anything publicly about Lisp while we were working on Viaweb. We never mentioned it to the press, and if you searched for Lisp on our Web site, all you'd find were the titles of two books in my bio. This was no accident. A startup should give its competitors as little information as possible. If they didn't know what language our software was written in, or didn't care, I wanted to keep it that way.[2]

The people who understood our technology best were the customers. They didn't care what language Viaweb was written in either, but they noticed that it worked really well. It let them build great looking online stores literally in minutes. And so, by word of mouth mostly, we got more and more users. By the end of 1996 we had about 70 stores online. At the end of 1997 we had 500. Six months later, when Yahoo bought us, we had 1070 users. Today, as Yahoo Store, this software continues to dominate its market. It's one of the more profitable pieces of Yahoo, and the stores built with it are the foundation of Yahoo Shopping. I left Yahoo in 1999, so I don't know exactly how many users they have now, but the last I heard there were about 20,000.

The Blub Paradox

What's so great about Lisp? And if Lisp is so great, why doesn't everyone use it? These sound like rhetorical questions, but actually they have straightforward answers. Lisp is so great not because of some magic quality visible only to devotees, but because it is simply the most powerful language available. And the reason everyone doesn't use it is that programming languages are not merely technologies, but habits of mind as well, and nothing changes slower. Of course, both these answers need explaining.

I'll begin with a shockingly controversial statement: programming languages vary in power.

Few would dispute, at least, that high level languages are more powerful than machine language. Most programmers today would agree that you do not, ordinarily, want to program in machine language. Instead, you should program in a high-level language, and have a compiler translate it into machine language for you. This idea is even built into the hardware now: since the 1980s, instruction

sets have been designed for compilers rather than human programmers.

Everyone knows it's a mistake to write your whole program by hand in machine language. What's less often understood is that there is a more general principle here: that if you have a choice of several languages, it is, all other things being equal, a mistake to program in anything but the most powerful one. [3]

There are many exceptions to this rule. If you're writing a program that has to work very closely with a program written in a certain language, it might be a good idea to write the new program in the same language. If you're writing a program that only has to do something very simple, like number crunching or bit manipulation, you may as well use a less abstract language, especially since it may be slightly faster. And if you're writing a short, throwaway program, you may be better off just using whatever language has the best library functions for the task. But in general, for application software, you want to be using the most powerful (reasonably efficient) language you can get, and using anything else is a mistake, of exactly the same kind, though possibly in a lesser degree, as programming in machine language.

You can see that machine language is very low level. But, at least as a kind of social convention, high-level languages are often all treated as equivalent. They're not. Technically the term "high-level language" doesn't mean anything very definite. There's no dividing line with machine languages on one side and all the high-level languages on the other. Languages fall along a continuum [4] of abstractness, from the most powerful all the way down to machine languages, which themselves vary in power.

Consider Cobol. Cobol is a high-level language, in the sense that it gets compiled into machine language. Would anyone seriously argue that Cobol is equivalent in power to, say, Python? It's probably closer to machine language than Python.

Or how about Perl 4? Between Perl 4 and Perl 5, lexical closures got added to the language. Most Perl hackers would agree that Perl 5 is more powerful than Perl 4. But once you've admitted that, you've admitted that one high level language can be more powerful than another. And it follows inexorably that, except in special cases, you ought to use the most powerful you can get.

This idea is rarely followed to its conclusion, though. After a certain age, programmers rarely switch languages voluntarily. Whatever language people happen to be used to, they tend to consider just good enough.

Programmers get very attached to their favorite languages, and I don't want to hurt anyone's feelings, so to explain this point I'm going to use a hypothetical language called Blub. Blub falls right in the middle of the abstractness continuum. It is not the most powerful language, but it is more powerful than Cobol or machine language.

And in fact, our hypothetical Blub programmer wouldn't use either of them. Of

course he wouldn't program in machine language. That's what compilers are for. And as for Cobol, he doesn't know how anyone can get anything done with it. It doesn't even have x (Blub feature of your choice).

As long as our hypothetical Blub programmer is looking down the power continuum, he knows he's looking down. Languages less powerful than Blub are obviously less powerful, because they're missing some feature he's used to. But when our hypothetical Blub programmer looks in the other direction, up the power continuum, he doesn't realize he's looking up. What he sees are merely weird languages. He probably considers them about equivalent in power to Blub, but with all this other hairy stuff thrown in as well. Blub is good enough for him, because he thinks in Blub.

When we switch to the point of view of a programmer using any of the languages higher up the power continuum, however, we find that he in turn looks down upon Blub. How can you get anything done in Blub? It doesn't even have y.

By induction, the only programmers in a position to see all the differences in power between the various languages are those who understand the most powerful one. (This is probably what Eric Raymond meant about Lisp making you a better programmer.) You can't trust the opinions of the others, because of the Blub paradox: they're satisfied with whatever language they happen to use, because it dictates the way they think about programs.

I know this from my own experience, as a high school kid writing programs in Basic. That language didn't even support recursion. It's hard to imagine writing programs without using recursion, but I didn't miss it at the time. I thought in Basic. And I was a whiz at it. Master of all I surveyed.

The five languages that Eric Raymond recommends to hackers fall at various points on the power continuum. Where they fall relative to one another is a sensitive topic. What I will say is that I think Lisp is at the top. And to support this claim I'll tell you about one of the things I find missing when I look at the other four languages. How can you get anything done in them, I think, without macros? [5]

Many languages have something called a macro. But Lisp macros are unique. And believe it or not, what they do is related to the parentheses. The designers of Lisp didn't put all those parentheses in the language just to be different. To the Blub programmer, Lisp code looks weird. But those parentheses are there for a reason. They are the outward evidence of a fundamental difference between Lisp and other languages.

Lisp code is made out of Lisp data objects. And not in the trivial sense that the source files contain characters, and strings are one of the data types supported by the language. Lisp code, after it's read by the parser, is made of data structures that you can traverse.

If you understand how compilers work, what's really going on is not so much that

Lisp has a strange syntax as that Lisp has no syntax. You write programs in the parse trees that get generated within the compiler when other languages are parsed. But these parse trees are fully accessible to your programs. You can write programs that manipulate them. In Lisp, these programs are called macros. They are programs that write programs.

Programs that write programs? When would you ever want to do that? Not very often, if you think in Cobol. All the time, if you think in Lisp. It would be convenient here if I could give an example of a powerful macro, and say there! how about that? But if I did, it would just look like gibberish to someone who didn't know Lisp; there isn't room here to explain everything you'd need to know to understand what it meant. In Ansi Common Lisp I tried to move things along as fast as I could, and even so I didn't get to macros until page 160.

But I think I can give a kind of argument that might be convincing. The source code of the Viaweb editor was probably about 20-25% macros. Macros are harder to write than ordinary Lisp functions, and it's considered to be bad style to use them when they're not necessary. So every macro in that code is there because it has to be. What that means is that at least 20-25% of the code in this program is doing things that you can't easily do in any other language. However skeptical the Blub programmer might be about my claims for the mysterious powers of Lisp, this ought to make him curious. We weren't writing this code for our own amusement. We were a tiny startup, programming as hard as we could in order to put technical barriers between us and our competitors.

A suspicious person might begin to wonder if there was some correlation here. A big chunk of our code was doing things that are very hard to do in other languages. The resulting software did things our competitors' software couldn't do. Maybe there was some kind of connection. I encourage you to follow that thread. There may be more to that old man hobbling along on his crutches than meets the eye.

Aikido for Startups

But I don't expect to convince anyone (over 25) to go out and learn Lisp. The purpose of this article is not to change anyone's mind, but to reassure people already interested in using Lisp-- people who know that Lisp is a powerful language, but worry because it isn't widely used. In a competitive situation, that's an advantage. Lisp's power is multiplied by the fact that your competitors don't get it.

If you think of using Lisp in a startup, you shouldn't worry that it isn't widely understood. You should hope that it stays that way. And it's likely to. It's the nature of programming languages to make most people satisfied with whatever they currently use. Computer hardware changes so much faster than personal habits that programming practice is usually ten to twenty years behind the processor. At places like MIT they were writing programs in high-level languages in the early 1960s, but many companies continued to write code in machine language

well into the 1980s. I bet a lot of people continued to write machine language until the processor, like a bartender eager to close up and go home, finally kicked them out by switching to a risc instruction set.

Ordinarily technology changes fast. But programming languages are different: programming languages are not just technology, but what programmers think in. They're half technology and half religion.[6] And so the median language, meaning whatever language the median programmer uses, moves as slow as an iceberg. Garbage collection, introduced by Lisp in about 1960, is now widely considered to be a good thing. Runtime typing, ditto, is growing in popularity. Lexical closures, introduced by Lisp in the early 1970s, are now, just barely, on the radar screen. Macros, introduced by Lisp in the mid 1960s, are still terra incognita.

Obviously, the median language has enormous momentum. I'm not proposing that you can fight this powerful force. What I'm proposing is exactly the opposite: that, like a practitioner of Aikido, you can use it against your opponents.

If you work for a big company, this may not be easy. You will have a hard time convincing the pointy-haired boss to let you build things in Lisp, when he has just read in the paper that some other language is poised, like Ada was twenty years ago, to take over the world. But if you work for a startup that doesn't have pointy-haired bosses yet, you can, like we did, turn the Blub paradox to your advantage: you can use technology that your competitors, glued immovably to the median language, will never be able to match.

If you ever do find yourself working for a startup, here's a handy tip for evaluating competitors. Read their job listings. Everything else on their site may be stock photos or the prose equivalent, but the job listings have to be specific about what they want, or they'll get the wrong candidates.

During the years we worked on Viaweb I read a lot of job descriptions. A new competitor seemed to emerge out of the woodwork every month or so. The first thing I would do, after checking to see if they had a live online demo, was look at their job listings. After a couple years of this I could tell which companies to worry about and which not to. The more of an IT flavor the job descriptions had, the less dangerous the company was. The safest kind were the ones that wanted Oracle experience. You never had to worry about those. You were also safe if they said they wanted C++ or Java developers. If they wanted Perl or Python programmers, that would be a bit frightening-- that's starting to sound like a company where the technical side, at least, is run by real hackers. If I had ever seen a job posting looking for Lisp hackers, I would have been really worried.

Notes

[1] Viaweb at first had two parts: the editor, written in Lisp, which people used to build their sites, and the ordering system, written in C, which handled orders. The

first version was mostly Lisp, because the ordering system was small. Later we added two more modules, an image generator written in C, and a back-office manager written mostly in Perl.

In January 2003, Yahoo released a new version of the editor written in C++ and Perl. It's hard to say whether the program is no longer written in Lisp, though, because to translate this program into C++ they literally had to write a Lisp interpreter: the source files of all the page-generating templates are still, as far as I know, Lisp code. (See Greenspun's Tenth Rule.)

[2] Robert Morris says that I didn't need to be secretive, because even if our competitors had known we were using Lisp, they wouldn't have understood why: "If they were that smart they'd already be programming in Lisp."

[3] All languages are equally powerful in the sense of being Turing equivalent, but that's not the sense of the word programmers care about. (No one wants to program a Turing machine.) The kind of power programmers care about may not be formally definable, but one way to explain it would be to say that it refers to features you could only get in the less powerful language by writing an interpreter for the more powerful language in it. If language A has an operator for removing spaces from strings and language B doesn't, that probably doesn't make A more powerful, because you can probably write a subroutine to do it in B. But if A supports, say, recursion, and B doesn't, that's not likely to be something you can fix by writing library functions.

[4] Note to nerds: or possibly a lattice, narrowing toward the top; it's not the shape that matters here but the idea that there is at least a partial order.

[5] It is a bit misleading to treat macros as a separate feature. In practice their usefulness is greatly enhanced by other Lisp features like lexical closures and rest parameters.

[6] As a result, comparisons of programming languages either take the form of religious wars or undergraduate textbooks so determinedly neutral that they're really works of anthropology. People who value their peace, or want tenure, avoid the topic. But the question is only half a religious one; there is something there worth studying, especially if you want to design new languages.

More Technical Details

You'll find this essay and 14 others in ***Hackers & Painters***.

Java's Cover

April 2001

This essay developed out of conversations I've had with several other programmers about why Java smelled suspicious. It's not a critique of Java! It is a case study of hacker's radar.

Over time, hackers develop a nose for good (and bad) technology. I thought it might be interesting to try and write down what made Java seem suspect to me.

Some people who've read this think it's an interesting attempt to write about something that hasn't been written about before. Others say I will get in trouble for appearing to be writing about things I don't understand. So, just in case it does any good, let me clarify that I'm not writing here about Java (which I have never used) but about hacker's radar (which I have thought about a lot).

The aphorism "you can't tell a book by its cover" originated in the times when books were sold in plain cardboard covers, to be bound by each purchaser according to his own taste. In those days, you couldn't tell a book by its cover. But publishing has advanced since then: present-day publishers work hard to make the cover something you can tell a book by.

I spend a lot of time in bookshops and I feel as if I have by now learned to understand everything publishers mean to tell me about a book, and perhaps a bit more. The time I haven't spent in bookshops I've spent mostly in front of computers, and I feel as if I've learned, to some degree, to judge technology by its cover as well. It may be just luck, but I've saved myself from a few technologies that turned out to be real stinkers.

So far, Java seems like a stinker to me. I've never written a Java program, never more than glanced over reference books about it, but I have a hunch that it won't be a very successful language. I may turn out to be mistaken; making predictions about technology is a dangerous business. But for what it's worth, as a sort of time capsule, here's why I don't like the look of Java:

1. It has been so energetically hyped. Real standards don't have to be promoted. No one had to promote C, or Unix, or HTML. A real standard tends to be already

established by the time most people hear about it. On the hacker radar screen, Perl is as big as Java, or bigger, just on the strength of its own merits.

2. It's aimed low. In the original Java white paper, Gosling explicitly says Java was designed not to be too difficult for programmers used to C. It was designed to be another C++: C plus a few ideas taken from more advanced languages. Like the creators of sitcoms or junk food or package tours, Java's designers were consciously designing a product for people not as smart as them. Historically, languages designed for other people to use have been bad: Cobol, PL/I, Pascal, Ada, C++. The good languages have been those that were designed for their own creators: C, Perl, Smalltalk, Lisp.

3. It has ulterior motives. Someone once said that the world would be a better place if people only wrote books because they had something to say, rather than because they wanted to write a book. Likewise, the reason we hear about Java all the time is not because it has something to say about programming languages. We hear about Java as part of a plan by Sun to undermine Microsoft.

4. No one loves it. C, Perl, Python, Smalltalk, and Lisp programmers love their languages. I've never heard anyone say that they loved Java.

5. People are forced to use it. A lot of the people I know using Java are using it because they feel they have to. Either it's something they felt they had to do to get funded, or something they thought customers would want, or something they were told to do by management. These are smart people; if the technology was good, they'd have used it voluntarily.

6. It has too many cooks. The best programming languages have been developed by small groups. Java seems to be run by a committee. If it turns out to be a good language, it will be the first time in history that a committee has designed a good language.

7. It's bureaucratic. From what little I know about Java, there seem to be a lot of protocols for doing things. Really good languages aren't like that. They let you do what you want and get out of the way.

8. It's pseudo-hip. Sun now pretends that Java is a grassroots, open-source language effort like Perl or Python. This one just happens to be controlled by a giant company. So the language is likely to have the same drab clunkiness as anything else that comes out of a big company.

9. It's designed for large organizations. Large organizations have different aims from hackers. They want languages that are (believed to be) suitable for use by large teams of mediocre programmers-- languages with features that, like the speed limiters in U-Haul trucks, prevent fools from doing too much damage. Hackers don't like a language that talks down to them. Hackers just want power. Historically, languages designed for large organizations (PL/I, Ada) have lost, while hacker languages (C, Perl) have won. The reason: today's teenage hacker is

tomorrow's CTO.

10. The wrong people like it. The programmers I admire most are not, on the whole, captivated by Java. Who does like Java? Suits, who don't know one language from another, but know that they keep hearing about Java in the press; programmers at big companies, who are amazed to find that there is something even better than C++; and plug-and-chug undergrads, who are ready to like anything that might get them a job (will this be on the test?). These people's opinions change with every wind.

11. Its daddy is in a pinch. Sun's business model is being undermined on two fronts. Cheap Intel processors, of the same type used in desktop machines, are now more than fast enough for servers. And FreeBSD seems to be at least as good an OS for servers as Solaris. Sun's advertising implies that you need Sun servers for industrial strength applications. If this were true, Yahoo would be first in line to buy Suns; but when I worked there, the servers were all Intel boxes running FreeBSD. This bodes ill for Sun's future. If Sun runs into trouble, they could drag Java down with them.

12. The DoD likes it. The Defense Department is encouraging developers to use Java. This seems to me the most damning sign of all. The Defense Department does a fine (though expensive) job of defending the country, but they love plans and procedures and protocols. Their culture is the opposite of hacker culture; on questions of software they will tend to bet wrong. The last time the DoD really liked a programming language, it was Ada.

Bear in mind, this is not a critique of Java, but a critique of its cover. I don't know Java well enough to like it or dislike it. This is just an explanation of why I don't find that I'm eager to learn it.

It may seem cavalier to dismiss a language before you've even tried writing programs in it. But this is something all programmers have to do. There are too many technologies out there to learn them all. You have to learn to judge by outward signs which will be worth your time. I have likewise cavalierly dismissed Cobol, Ada, Visual Basic, the IBM AS400, VRML, ISO 9000, the SET protocol, VMS, Novell Netware, and CORBA, among others. They just smelled wrong.

It could be that in Java's case I'm mistaken. It could be that a language promoted by one big company to undermine another, designed by a committee for a "mainstream" audience, hyped to the skies, and beloved of the DoD, happens nonetheless to be a clean, beautiful, powerful language that I would love programming in. It could be, but it seems very unlikely.

Being Popular

May 2001

(This article was written as a kind of business plan for a [new language](). So it is missing (because it takes for granted) the most important feature of a good programming language: very powerful abstractions.)

A friend of mine once told an eminent operating systems expert that he wanted to design a really good programming language. The expert told him that it would be a waste of time, that programming languages don't become popular or unpopular based on their merits, and so no matter how good his language was, no one would use it. At least, that was what had happened to the language *he* had designed.

What does make a language popular? Do popular languages deserve their popularity? Is it worth trying to define a good programming language? How would you do it?

I think the answers to these questions can be found by looking at hackers, and learning what they want. Programming languages are *for* hackers, and a programming language is good as a programming language (rather than, say, an exercise in denotational semantics or compiler design) if and only if hackers like it.

1 The Mechanics of Popularity

It's true, certainly, that most people don't choose programming languages simply based on their merits. Most programmers are told what language to use by someone else. And yet I think the effect of such external factors on the popularity of programming languages is not as great as it's sometimes thought to be. I think a bigger problem is that a hacker's idea of a good programming language is not the same as most language designers'.

Between the two, the hacker's opinion is the one that matters. Programming languages are not theorems. They're tools, designed for people, and they have to be designed to suit human strengths and weaknesses as much as shoes have to be designed for human feet. If a shoe pinches when you put it on, it's a bad shoe, however elegant it may be as a piece of sculpture.

It may be that the majority of programmers can't tell a good language from a bad one. But that's no different with any other tool. It doesn't mean that it's a waste of

time to try designing a good language. [Expert hackers](#) can tell a good language when they see one, and they'll use it. Expert hackers are a tiny minority, admittedly, but that tiny minority write all the good software, and their influence is such that the rest of the programmers will tend to use whatever language they use. Often, indeed, it is not merely influence but command: often the expert hackers are the very people who, as their bosses or faculty advisors, tell the other programmers what language to use.

The opinion of expert hackers is not the only force that determines the relative popularity of programming languages — legacy software (Cobol) and hype (Ada, Java) also play a role — but I think it is the most powerful force over the long term. Given an initial critical mass and enough time, a programming language probably becomes about as popular as it deserves to be. And popularity further separates good languages from bad ones, because feedback from real live users always leads to improvements. Look at how much any popular language has changed during its life. Perl and Fortran are extreme cases, but even Lisp has changed a lot. Lisp 1.5 didn't have macros, for example; these evolved later, after hackers at MIT had spent a couple years using Lisp to write real programs. [1]

So whether or not a language has to be good to be popular, I think a language has to be popular to be good. And it has to stay popular to stay good. The state of the art in programming languages doesn't stand still. And yet the Lisps we have today are still pretty much what they had at MIT in the mid-1980s, because that's the last time Lisp had a sufficiently large and demanding user base.

Of course, hackers have to know about a language before they can use it. How are they to hear? From other hackers. But there has to be some initial group of hackers using the language for others even to hear about it. I wonder how large this group has to be; how many users make a critical mass? Off the top of my head, I'd say twenty. If a language had twenty separate users, meaning twenty users who decided on their own to use it, I'd consider it to be real.

Getting there can't be easy. I would not be surprised if it is harder to get from zero to twenty than from twenty to a thousand. The best way to get those initial twenty users is probably to use a trojan horse: to give people an application they want, which happens to be written in the new language.

2 External Factors

Let's start by acknowledging one external factor that does affect the popularity of a programming language. To become popular, a programming language has to be the scripting language of a popular system. Fortran and Cobol were the scripting languages of early IBM mainframes. C was the scripting language of Unix, and so, later, was Perl. Tcl is the scripting language of Tk. Java and Javascript are intended to be the scripting languages of web browsers.

Lisp is not a massively popular language because it is not the scripting language of a massively popular system. What popularity it retains dates back to the 1960s

and 1970s, when it was the scripting language of MIT. A lot of the great programmers of the day were associated with MIT at some point. And in the early 1970s, before C, MIT's dialect of Lisp, called MacLisp, was one of the only programming languages a serious hacker would want to use.

Today Lisp is the scripting language of two moderately popular systems, Emacs and Autocad, and for that reason I suspect that most of the Lisp programming done today is done in Emacs Lisp or AutoLisp.

Programming languages don't exist in isolation. To hack is a transitive verb — hackers are usually hacking something — and in practice languages are judged relative to whatever they're used to hack. So if you want to design a popular language, you either have to supply more than a language, or you have to design your language to replace the scripting language of some existing system.

Common Lisp is unpopular partly because it's an orphan. It did originally come with a system to hack: the Lisp Machine. But Lisp Machines (along with parallel computers) were steamrollered by the increasing power of general purpose processors in the 1980s. Common Lisp might have remained popular if it had been a good scripting language for Unix. It is, alas, an atrociously bad one.

One way to describe this situation is to say that a language isn't judged on its own merits. Another view is that a programming language really isn't a programming language unless it's also the scripting language of something. This only seems unfair if it comes as a surprise. I think it's no more unfair than expecting a programming language to have, say, an implementation. It's just part of what a programming language is.

A programming language does need a good implementation, of course, and this must be free. Companies will pay for software, but individual hackers won't, and it's the hackers you need to attract.

A language also needs to have a book about it. The book should be thin, well-written, and full of good examples. K&R is the ideal here. At the moment I'd almost say that a language has to have a book published by O'Reilly. That's becoming the test of mattering to hackers.

There should be online documentation as well. In fact, the book can start as online documentation. But I don't think that physical books are outmoded yet. Their format is convenient, and the de facto censorship imposed by publishers is a useful if imperfect filter. Bookstores are one of the most important places for learning about new languages.

3 Brevity

Given that you can supply the three things any language needs — a free implementation, a book, and something to hack — how do you make a language that hackers will like?

One thing hackers like is brevity. Hackers are lazy, in the same way that mathematicians and modernist architects are lazy: they hate anything extraneous. It would not be far from the truth to say that a hacker about to write a program decides what language to use, at least subconsciously, based on the total number of characters he'll have to type. If this isn't precisely how hackers think, a language designer would do well to act as if it were.

It is a mistake to try to baby the user with long-winded expressions that are meant to resemble English. Cobol is notorious for this flaw. A hacker would consider being asked to write

add x to y giving z

instead of

z = x+y

as something between an insult to his intelligence and a sin against God.

It has sometimes been said that Lisp should use first and rest instead of car and cdr, because it would make programs easier to read. Maybe for the first couple hours. But a hacker can learn quickly enough that car means the first element of a list and cdr means the rest. Using first and rest means 50% more typing. And they are also different lengths, meaning that the arguments won't line up when they're called, as car and cdr often are, in successive lines. I've found that it matters a lot how code lines up on the page. I can barely read Lisp code when it is set in a variable-width font, and friends say this is true for other languages too.

Brevity is one place where strongly typed languages lose. All other things being equal, no one wants to begin a program with a bunch of declarations. Anything that can be implicit, should be.

The individual tokens should be short as well. Perl and Common Lisp occupy opposite poles on this question. Perl programs can be almost cryptically dense, while the names of built-in Common Lisp operators are comically long. The designers of Common Lisp probably expected users to have text editors that would type these long names for them. But the cost of a long name is not just the cost of typing it. There is also the cost of reading it, and the cost of the space it takes up on your screen.

4 Hackability

There is one thing more important than brevity to a hacker: being able to do what you want. In the history of programming languages a surprising amount of effort has gone into preventing programmers from doing things considered to be improper. This is a dangerously presumptuous plan. How can the language designer know what the programmer is going to need to do? I think language

designers would do better to consider their target user to be a genius who will need to do things they never anticipated, rather than a bumbler who needs to be protected from himself. The bumbler will shoot himself in the foot anyway. You may save him from referring to variables in another package, but you can't save him from writing a badly designed program to solve the wrong problem, and taking forever to do it.

Good programmers often want to do dangerous and unsavory things. By unsavory I mean things that go behind whatever semantic facade the language is trying to present: getting hold of the internal representation of some high-level abstraction, for example. Hackers like to hack, and hacking means getting inside things and second guessing the original designer.

Let yourself be second guessed. When you make any tool, people use it in ways you didn't intend, and this is especially true of a highly articulated tool like a programming language. Many a hacker will want to tweak your semantic model in a way that you never imagined. I say, let them; give the programmer access to as much internal stuff as you can without endangering runtime systems like the garbage collector.

In Common Lisp I have often wanted to iterate through the fields of a struct — to comb out references to a deleted object, for example, or find fields that are uninitialized. I know the structs are just vectors underneath. And yet I can't write a general purpose function that I can call on any struct. I can only access the fields by name, because that's what a struct is supposed to mean.

A hacker may only want to subvert the intended model of things once or twice in a big program. But what a difference it makes to be able to. And it may be more than a question of just solving a problem. There is a kind of pleasure here too. Hackers share the surgeon's secret pleasure in poking about in gross innards, the teenager's secret pleasure in popping zits. [2] For boys, at least, certain kinds of horrors are fascinating. Maxim magazine publishes an annual volume of photographs, containing a mix of pin-ups and grisly accidents. They know their audience.

Historically, Lisp has been good at letting hackers have their way. The political correctness of Common Lisp is an aberration. Early Lisps let you get your hands on everything. A good deal of that spirit is, fortunately, preserved in macros. What a wonderful thing, to be able to make arbitrary transformations on the source code.

Classic macros are a real hacker's tool — simple, powerful, and dangerous. It's so easy to understand what they do: you call a function on the macro's arguments, and whatever it returns gets inserted in place of the macro call. Hygienic macros embody the opposite principle. They try to protect you from understanding what they're doing. I have never heard hygienic macros explained in one sentence. And they are a classic example of the dangers of deciding what programmers are allowed to want. Hygienic macros are intended to protect me from variable capture, among other things, but variable capture is exactly what I want in some

macros.

A really good language should be both clean and dirty: cleanly designed, with a small core of well understood and highly orthogonal operators, but dirty in the sense that it lets hackers have their way with it. C is like this. So were the early Lisps. A real hacker's language will always have a slightly raffish character.

A good programming language should have features that make the kind of people who use the phrase "software engineering" shake their heads disapprovingly. At the other end of the continuum are languages like Ada and Pascal, models of propriety that are good for teaching and not much else.

5 Throwaway Programs

To be attractive to hackers, a language must be good for writing the kinds of programs they want to write. And that means, perhaps surprisingly, that it has to be good for writing throwaway programs.

A throwaway program is a program you write quickly for some limited task: a program to automate some system administration task, or generate test data for a simulation, or convert data from one format to another. The surprising thing about throwaway programs is that, like the "temporary" buildings built at so many American universities during World War II, they often don't get thrown away. Many evolve into real programs, with real features and real users.

I have a hunch that the best big programs begin life this way, rather than being designed big from the start, like the Hoover Dam. It's terrifying to build something big from scratch. When people take on a project that's too big, they become overwhelmed. The project either gets bogged down, or the result is sterile and wooden: a shopping mall rather than a real downtown, Brasilia rather than Rome, Ada rather than C.

Another way to get a big program is to start with a throwaway program and keep improving it. This approach is less daunting, and the design of the program benefits from evolution. I think, if one looked, that this would turn out to be the way most big programs were developed. And those that did evolve this way are probably still written in whatever language they were first written in, because it's rare for a program to be ported, except for political reasons. And so, paradoxically, if you want to make a language that is used for big systems, you have to make it good for writing throwaway programs, because that's where big systems come from.

Perl is a striking example of this idea. It was not only designed for writing throwaway programs, but was pretty much a throwaway program itself. Perl began life as a collection of utilities for generating reports, and only evolved into a programming language as the throwaway programs people wrote in it grew larger. It was not until Perl 5 (if then) that the language was suitable for writing serious programs, and yet it was already massively popular.

What makes a language good for throwaway programs? To start with, it must be readily available. A throwaway program is something that you expect to write in an hour. So the language probably must already be installed on the computer you're using. It can't be something you have to install before you use it. It has to be there. C was there because it came with the operating system. Perl was there because it was originally a tool for system administrators, and yours had already installed it.

Being available means more than being installed, though. An interactive language, with a command-line interface, is more available than one that you have to compile and run separately. A popular programming language should be interactive, and start up fast.

Another thing you want in a throwaway program is brevity. Brevity is always attractive to hackers, and never more so than in a program they expect to turn out in an hour.

6 Libraries

Of course the ultimate in brevity is to have the program already written for you, and merely to call it. And this brings us to what I think will be an increasingly important feature of programming languages: library functions. Perl wins because it has large libraries for manipulating strings. This class of library functions are especially important for throwaway programs, which are often originally written for converting or extracting data. Many Perl programs probably begin as just a couple library calls stuck together.

I think a lot of the advances that happen in programming languages in the next fifty years will have to do with library functions. I think future programming languages will have libraries that are as carefully designed as the core language. Programming language design will not be about whether to make your language strongly or weakly typed, or object oriented, or functional, or whatever, but about how to design great libraries. The kind of language designers who like to think about how to design type systems may shudder at this. It's almost like writing applications! Too bad. Languages are for programmers, and libraries are what programmers need.

It's hard to design good libraries. It's not simply a matter of writing a lot of code. Once the libraries get too big, it can sometimes take longer to find the function you need than to write the code yourself. Libraries need to be designed using a small set of orthogonal operators, just like the core language. It ought to be possible for the programmer to guess what library call will do what he needs.

Libraries are one place Common Lisp falls short. There are only rudimentary libraries for manipulating strings, and almost none for talking to the operating system. For historical reasons, Common Lisp tries to pretend that the OS doesn't exist. And because you can't talk to the OS, you're unlikely to be able to write a

serious program using only the built-in operators in Common Lisp. You have to use some implementation-specific hacks as well, and in practice these tend not to give you everything you want. Hackers would think a lot more highly of Lisp if Common Lisp had powerful string libraries and good OS support.

7 Syntax

Could a language with Lisp's syntax, or more precisely, lack of syntax, ever become popular? I don't know the answer to this question. I do think that syntax is not the main reason Lisp isn't currently popular. Common Lisp has worse problems than unfamiliar syntax. I know several programmers who are comfortable with prefix syntax and yet use Perl by default, because it has powerful string libraries and can talk to the os.

There are two possible problems with prefix notation: that it is unfamiliar to programmers, and that it is not dense enough. The conventional wisdom in the Lisp world is that the first problem is the real one. I'm not so sure. Yes, prefix notation makes ordinary programmers panic. But I don't think ordinary programmers' opinions matter. Languages become popular or unpopular based on what expert hackers think of them, and I think expert hackers might be able to deal with prefix notation. Perl syntax can be pretty incomprehensible, but that has not stood in the way of Perl's popularity. If anything it may have helped foster a Perl cult.

A more serious problem is the diffuseness of prefix notation. For expert hackers, that really is a problem. No one wants to write (aref a x y) when they could write a[x,y].

In this particular case there is a way to finesse our way out of the problem. If we treat data structures as if they were functions on indexes, we could write (a x y) instead, which is even shorter than the Perl form. Similar tricks may shorten other types of expressions.

We can get rid of (or make optional) a lot of parentheses by making indentation significant. That's how programmers read code anyway: when indentation says one thing and delimiters say another, we go by the indentation. Treating indentation as significant would eliminate this common source of bugs as well as making programs shorter.

Sometimes infix syntax is easier to read. This is especially true for math expressions. I've used Lisp my whole programming life and I still don't find prefix math expressions natural. And yet it is convenient, especially when you're generating code, to have operators that take any number of arguments. So if we do have infix syntax, it should probably be implemented as some kind of read-macro.

I don't think we should be religiously opposed to introducing syntax into Lisp, as long as it translates in a well-understood way into underlying s-expressions. There

is already a good deal of syntax in Lisp. It's not necessarily bad to introduce more, as long as no one is forced to use it. In Common Lisp, some delimiters are reserved for the language, suggesting that at least some of the designers intended to have more syntax in the future.

One of the most egregiously unlispy pieces of syntax in Common Lisp occurs in format strings; format is a language in its own right, and that language is not Lisp. If there were a plan for introducing more syntax into Lisp, format specifiers might be able to be included in it. It would be a good thing if macros could generate format specifiers the way they generate any other kind of code.

An eminent Lisp hacker told me that his copy of CLTL falls open to the section format. Mine too. This probably indicates room for improvement. It may also mean that programs do a lot of I/O.

8 Efficiency

A good language, as everyone knows, should generate fast code. But in practice I don't think fast code comes primarily from things you do in the design of the language. As Knuth pointed out long ago, speed only matters in certain critical bottlenecks. And as many programmers have observed since, one is very often mistaken about where these bottlenecks are.

So, in practice, the way to get fast code is to have a very good profiler, rather than by, say, making the language strongly typed. You don't need to know the type of every argument in every call in the program. You do need to be able to declare the types of arguments in the bottlenecks. And even more, you need to be able to find out where the bottlenecks are.

One complaint people have had with Lisp is that it's hard to tell what's expensive. This might be true. It might also be inevitable, if you want to have a very abstract language. And in any case I think good profiling would go a long way toward fixing the problem: you'd soon learn what was expensive.

Part of the problem here is social. Language designers like to write fast compilers. That's how they measure their skill. They think of the profiler as an add-on, at best. But in practice a good profiler may do more to improve the speed of actual programs written in the language than a compiler that generates fast code. Here, again, language designers are somewhat out of touch with their users. They do a really good job of solving slightly the wrong problem.

It might be a good idea to have an active profiler — to push performance data to the programmer instead of waiting for him to come asking for it. For example, the editor could display bottlenecks in red when the programmer edits the source code. Another approach would be to somehow represent what's happening in running programs. This would be an especially big win in server-based applications, where you have lots of running programs to look at. An active profiler could show graphically what's happening in memory as a program's running, or

even make sounds that tell what's happening.

Sound is a good cue to problems. In one place I worked, we had a big board of dials showing what was happening to our web servers. The hands were moved by little servomotors that made a slight noise when they turned. I couldn't see the board from my desk, but I found that I could tell immediately, by the sound, when there was a problem with a server.

It might even be possible to write a profiler that would automatically detect inefficient algorithms. I would not be surprised if certain patterns of memory access turned out to be sure signs of bad algorithms. If there were a little guy running around inside the computer executing our programs, he would probably have as long and plaintive a tale to tell about his job as a federal government employee. I often have a feeling that I'm sending the processor on a lot of wild goose chases, but I've never had a good way to look at what it's doing.

A number of Lisps now compile into byte code, which is then executed by an interpreter. This is usually done to make the implementation easier to port, but it could be a useful language feature. It might be a good idea to make the byte code an official part of the language, and to allow programmers to use inline byte code in bottlenecks. Then such optimizations would be portable too.

The nature of speed, as perceived by the end-user, may be changing. With the rise of server-based applications, more and more programs may turn out to be i/o-bound. It will be worth making i/o fast. The language can help with straightforward measures like simple, fast, formatted output functions, and also with deep structural changes like caching and persistent objects.

Users are interested in response time. But another kind of efficiency will be increasingly important: the number of simultaneous users you can support per processor. Many of the interesting applications written in the near future will be server-based, and the number of users per server is the critical question for anyone hosting such applications. In the capital cost of a business offering a server-based application, this is the divisor.

For years, efficiency hasn't mattered much in most end-user applications. Developers have been able to assume that each user would have an increasingly powerful processor sitting on their desk. And by Parkinson's Law, software has expanded to use the resources available. That will change with server-based applications. In that world, the hardware and software will be supplied together. For companies that offer server-based applications, it will make a very big difference to the bottom line how many users they can support per server.

In some applications, the processor will be the limiting factor, and execution speed will be the most important thing to optimize. But often memory will be the limit; the number of simultaneous users will be determined by the amount of memory you need for each user's data. The language can help here too. Good support for threads will enable all the users to share a single heap. It may also help to have

persistent objects and/or language level support for lazy loading.

9 Time

The last ingredient a popular language needs is time. No one wants to write programs in a language that might go away, as so many programming languages do. So most hackers will tend to wait until a language has been around for a couple years before even considering using it.

Inventors of wonderful new things are often surprised to discover this, but you need time to get any message through to people. A friend of mine rarely does anything the first time someone asks him. He knows that people sometimes ask for things that they turn out not to want. To avoid wasting his time, he waits till the third or fourth time he's asked to do something; by then, whoever's asking him may be fairly annoyed, but at least they probably really do want whatever they're asking for.

Most people have learned to do a similar sort of filtering on new things they hear about. They don't even start paying attention until they've heard about something ten times. They're perfectly justified: the majority of hot new whatevers do turn out to be a waste of time, and eventually go away. By delaying learning VRML, I avoided having to learn it at all.

So anyone who invents something new has to expect to keep repeating their message for years before people will start to get it. We wrote what was, as far as I know, the first web-server based application, and it took us years to get it through to people that it didn't have to be downloaded. It wasn't that they were stupid. They just had us tuned out.

The good news is, simple repetition solves the problem. All you have to do is keep telling your story, and eventually people will start to hear. It's not when people notice you're there that they pay attention; it's when they notice you're still there.

It's just as well that it usually takes a while to gain momentum. Most technologies evolve a good deal even after they're first launched — programming languages especially. Nothing could be better, for a new techology, than a few years of being used only by a small number of early adopters. Early adopters are sophisticated and demanding, and quickly flush out whatever flaws remain in your technology. When you only have a few users you can be in close contact with all of them. And early adopters are forgiving when you improve your system, even if this causes some breakage.

There are two ways new technology gets introduced: the organic growth method, and the big bang method. The organic growth method is exemplified by the classic seat-of-the-pants underfunded garage startup. A couple guys, working in obscurity, develop some new technology. They launch it with no marketing and initially have only a few (fanatically devoted) users. They continue to improve the technology, and meanwhile their user base grows by word of mouth. Before they know it,

they're big.

The other approach, the big bang method, is exemplified by the VC-backed, heavily marketed startup. They rush to develop a product, launch it with great publicity, and immediately (they hope) have a large user base.

Generally, the garage guys envy the big bang guys. The big bang guys are smooth and confident and respected by the VCs. They can afford the best of everything, and the PR campaign surrounding the launch has the side effect of making them celebrities. The organic growth guys, sitting in their garage, feel poor and unloved. And yet I think they are often mistaken to feel sorry for themselves. Organic growth seems to yield better technology and richer founders than the big bang method. If you look at the dominant technologies today, you'll find that most of them grew organically.

This pattern doesn't only apply to companies. You see it in sponsored research too. Multics and Common Lisp were big-bang projects, and Unix and MacLisp were organic growth projects.

10 Redesign

"The best writing is rewriting," wrote E. B. White. Every good writer knows this, and it's true for software too. The most important part of design is redesign. Programming languages, especially, don't get redesigned enough.

To write good software you must simultaneously keep two opposing ideas in your head. You need the young hacker's naive faith in his abilities, and at the same time the veteran's skepticism. You have to be able to think how hard can it be? with one half of your brain while thinking it will never work with the other.

The trick is to realize that there's no real contradiction here. You want to be optimistic and skeptical about two different things. You have to be optimistic about the possibility of solving the problem, but skeptical about the value of whatever solution you've got so far.

People who do good work often think that whatever they're working on is no good. Others see what they've done and are full of wonder, but the creator is full of worry. This pattern is no coincidence: it is the worry that made the work good.

If you can keep hope and worry balanced, they will drive a project forward the same way your two legs drive a bicycle forward. In the first phase of the two-cycle innovation engine, you work furiously on some problem, inspired by your confidence that you'll be able to solve it. In the second phase, you look at what you've done in the cold light of morning, and see all its flaws very clearly. But as long as your critical spirit doesn't outweigh your hope, you'll be able to look at your admittedly incomplete system, and think, how hard can it be to get the rest of the way?, thereby continuing the cycle.

It's tricky to keep the two forces balanced. In young hackers, optimism predominates. They produce something, are convinced it's great, and never improve it. In old hackers, skepticism predominates, and they won't even dare to take on ambitious projects.

Anything you can do to keep the redesign cycle going is good. Prose can be rewritten over and over until you're happy with it. But software, as a rule, doesn't get redesigned enough. Prose has readers, but software has *users.* If a writer rewrites an essay, people who read the old version are unlikely to complain that their thoughts have been broken by some newly introduced incompatibility.

Users are a double-edged sword. They can help you improve your language, but they can also deter you from improving it. So choose your users carefully, and be slow to grow their number. Having users is like optimization: the wise course is to delay it. Also, as a general rule, you can at any given time get away with changing more than you think. Introducing change is like pulling off a bandage: the pain is a memory almost as soon as you feel it.

Everyone knows that it's not a good idea to have a language designed by a committee. Committees yield bad design. But I think the worst danger of committees is that they interfere with redesign. It is so much work to introduce changes that no one wants to bother. Whatever a committee decides tends to stay that way, even if most of the members don't like it.

Even a committee of two gets in the way of redesign. This happens particularly in the interfaces between pieces of software written by two different people. To change the interface both have to agree to change it at once. And so interfaces tend not to change at all, which is a problem because they tend to be one of the most ad hoc parts of any system.

One solution here might be to design systems so that interfaces are horizontal instead of vertical — so that modules are always vertically stacked strata of abstraction. Then the interface will tend to be owned by one of them. The lower of two levels will either be a language in which the upper is written, in which case the lower level will own the interface, or it will be a slave, in which case the interface can be dictated by the upper level.

11 Lisp

What all this implies is that there is hope for a new Lisp. There is hope for any language that gives hackers what they want, including Lisp. I think we may have made a mistake in thinking that hackers are turned off by Lisp's strangeness. This comforting illusion may have prevented us from seeing the real problem with Lisp, or at least Common Lisp, which is that it sucks for doing what hackers want to do. A hacker's language needs powerful libraries and something to hack. Common Lisp has neither. A hacker's language is terse and hackable. Common Lisp is not.

The good news is, it's not Lisp that sucks, but Common Lisp. If we can develop a

new Lisp that is a real hacker's language, I think hackers will use it. They will use whatever language does the job. All we have to do is make sure this new Lisp does some important job better than other languages.

History offers some encouragement. Over time, successive new programming languages have taken more and more features from Lisp. There is no longer much left to copy before the language you've made is Lisp. The latest hot language, Python, is a watered-down Lisp with infix syntax and no macros. A new Lisp would be a natural step in this progression.

I sometimes think that it would be a good marketing trick to call it an improved version of Python. That sounds hipper than Lisp. To many people, Lisp is a slow AI language with a lot of parentheses. Fritz Kunze's official biography carefully avoids mentioning the L-word. But my guess is that we shouldn't be afraid to call the new Lisp Lisp. Lisp still has a lot of latent respect among the very best hackers — the ones who took 6.001 and understood it, for example. And those are the users you need to win.

In "How to Become a Hacker," Eric Raymond describes Lisp as something like Latin or Greek — a language you should learn as an intellectual exercise, even though you won't actually use it:

> Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

If I didn't know Lisp, reading this would set me asking questions. A language that would make me a better programmer, if it means anything at all, means a language that would be better for programming. And that is in fact the implication of what Eric is saying.

As long as that idea is still floating around, I think hackers will be receptive enough to a new Lisp, even if it is called Lisp. But this Lisp must be a hacker's language, like the classic Lisps of the 1970s. It must be terse, simple, and hackable. And it must have powerful libraries for doing what hackers want to do now.

In the matter of libraries I think there is room to beat languages like Perl and Python at their own game. A lot of the new applications that will need to be written in the coming years will be [server-based applications](). There's no reason a new Lisp shouldn't have string libraries as good as Perl, and if this new Lisp also had powerful libraries for server-based applications, it could be very popular. Real hackers won't turn up their noses at a new tool that will let them solve hard problems with a few library calls. Remember, hackers are lazy.

It could be an even bigger win to have core language support for server-based applications. For example, explicit support for programs with multiple users, or data ownership at the level of type tags.

Server-based applications also give us the answer to the question of what this new Lisp will be used to hack. It would not hurt to make Lisp better as a scripting language for Unix. (It would be hard to make it worse.) But I think there are areas where existing languages would be easier to beat. I think it might be better to follow the model of Tcl, and supply the Lisp together with a complete system for supporting server-based applications. Lisp is a natural fit for server-based applications. Lexical closures provide a way to get the effect of subroutines when the ui is just a series of web pages. S-expressions map nicely onto html, and macros are good at generating it. There need to be better tools for writing server-based applications, and there needs to be a new Lisp, and the two would work very well together.

12 The Dream Language

By way of summary, let's try describing the hacker's dream language. The dream language is [beautiful](), clean, and terse. It has an interactive toplevel that starts up fast. You can write programs to solve common problems with very little code. Nearly all the code in any program you write is code that's specific to your application. Everything else has been done for you.

The syntax of the language is brief to a fault. You never have to type an unnecessary character, or even to use the shift key much.

Using big abstractions you can write the first version of a program very quickly. Later, when you want to optimize, there's a really good profiler that tells you where to focus your attention. You can make inner loops blindingly fast, even writing inline byte code if you need to.

There are lots of good examples to learn from, and the language is intuitive enough that you can learn how to use it from examples in a couple minutes. You don't need to look in the manual much. The manual is thin, and has few warnings and qualifications.

The language has a small core, and powerful, highly orthogonal libraries that are as carefully designed as the core language. The libraries all work well together; everything in the language fits together like the parts in a fine camera. Nothing is deprecated, or retained for compatibility. The source code of all the libraries is readily available. It's easy to talk to the operating system and to applications written in other languages.

The language is built in layers. The higher-level abstractions are built in a very transparent way out of lower-level abstractions, which you can get hold of if you want.

Nothing is hidden from you that doesn't absolutely have to be. The language offers abstractions only as a way of saving you work, rather than as a way of telling you what to do. In fact, the language encourages you to be an equal participant in its design. You can change everything about it, including even its syntax, and

anything you write has, as much as possible, the same status as what comes predefined.

Notes

[1] Macros very close to the modern idea were proposed by Timothy Hart in 1964, two years after Lisp 1.5 was released. What was missing, initially, were ways to avoid variable capture and multiple evaluation; Hart's examples are subject to both.

[2] In *When the Air Hits Your Brain,* neurosurgeon Frank Vertosick recounts a conversation in which his chief resident, Gary, talks about the difference between surgeons and internists ("fleas"):

> Gary and I ordered a large pizza and found an open booth. The chief lit a cigarette. "Look at those goddamn fleas, jabbering about some disease they'll see once in their lifetimes. That's the trouble with fleas, they only like the bizarre stuff. They hate their bread and butter cases. That's the difference between us and the fucking fleas. See, we love big juicy lumbar disc herniations, but they hate hypertension...."

It's hard to think of a lumbar disc herniation as juicy (except literally). And yet I think I know what they mean. I've often had a juicy bug to track down. Someone who's not a programmer would find it hard to imagine that there could be pleasure in a bug. Surely it's better if everything just works. In one way, it is. And yet there is undeniably a grim satisfaction in hunting down certain sorts of bugs.

[Postscript Version](#)

[Arc](#)

[Five Questions about Language Design](#)

[How to Become a Hacker](#)

.

[Japanese Translation](#)

Five Questions about Language Design

May 2001

(These are some notes I made for a panel discussion on programming language design at MIT on May 10, 2001.)

GUIDING PHILOSOPHY

1. Programming Languages Are for People.

Programming languages are how people talk to computers. The computer would be just as happy speaking any language that was unambiguous. The reason we have high level languages is because people can't deal with machine language. The point of programming languages is to prevent our poor frail human brains from being overwhelmed by a mass of detail.

Architects know that some kinds of design problems are more personal than others. One of the cleanest, most abstract design problems is designing bridges. There your job is largely a matter of spanning a given distance with the least material. The other end of the spectrum is designing chairs. Chair designers have to spend their time thinking about human butts.

Software varies in the same way. Designing algorithms for routing data through a network is a nice, abstract problem, like designing bridges. Whereas designing programming languages is like designing chairs: it's all about dealing with human weaknesses.

Most of us hate to acknowledge this. Designing systems of great mathematical elegance sounds a lot more appealing to most of us than pandering to human weaknesses. And there is a role for mathematical elegance: some kinds of elegance make programs easier to understand. But elegance is not an end in itself.

And when I say languages have to be designed to suit human weaknesses, I don't mean that languages have to be designed for bad programmers. In fact I think you ought to design for the [best programmers](#), but even the best programmers have limitations. I don't think anyone would like programming in a language where all the variables were the letter x with integer subscripts.

2. Design for Yourself and Your Friends.

If you look at the history of programming languages, a lot of the best ones were languages designed for their own authors to use, and a lot of the worst ones were

designed for other people to use.

When languages are designed for other people, it's always a specific group of other people: people not as smart as the language designer. So you get a language that talks down to you. Cobol is the most extreme case, but a lot of languages are pervaded by this spirit.

It has nothing to do with how abstract the language is. C is pretty low-level, but it was designed for its authors to use, and that's why hackers like it.

The argument for designing languages for bad programmers is that there are more bad programmers than good programmers. That may be so. But those few good programmers write a disproportionately large percentage of the software.

I'm interested in the question, how do you design a language that the very best hackers will like? I happen to think this is identical to the question, how do you design a good programming language?, but even if it isn't, it is at least an interesting question.

3. Give the Programmer as Much Control as Possible.

Many languages (especially the ones designed for other people) have the attitude of a governess: they try to prevent you from doing things that they think aren't good for you. I like the opposite approach: give the programmer as much control as you can.

When I first learned Lisp, what I liked most about it was that it considered me an equal partner. In the other languages I had learned up till then, there was the language and there was my program, written in the language, and the two were very separate. But in Lisp the functions and macros I wrote were just like those that made up the language itself. I could rewrite the language if I wanted. It had the same appeal as open-source software.

4. Aim for Brevity.

Brevity is underestimated and even scorned. But if you look into the hearts of hackers, you'll see that they really love it. How many times have you heard hackers speak fondly of how in, say, APL, they could do amazing things with just a couple lines of code? I think anything that really smart people really love is worth paying attention to.

I think almost anything you can do to make programs shorter is good. There should be lots of library functions; anything that can be implicit should be; the syntax should be terse to a fault; even the names of things should be short.

And it's not only programs that should be short. The manual should be thin as well. A good part of manuals is taken up with clarifications and reservations and warnings and special cases. If you force yourself to shorten the manual, in the best case you do it by fixing the things in the language that required so much explanation.

5. Admit What Hacking Is.

A lot of people wish that hacking was mathematics, or at least something like a

natural science. I think hacking is more like architecture. Architecture is related to physics, in the sense that architects have to design buildings that don't fall down, but the actual goal of architects is to make great buildings, not to make discoveries about statics.

What hackers like to do is make great programs. And I think, at least in our own minds, we have to remember that it's an admirable thing to write great programs, even when this work doesn't translate easily into the conventional intellectual currency of research papers. Intellectually, it is just as worthwhile to design a language programmers will love as it is to design a horrible one that embodies some idea you can publish a paper about.

OPEN PROBLEMS

1. How to Organize Big Libraries?

Libraries are becoming an increasingly important component of programming languages. They're also getting bigger, and this can be dangerous. If it takes longer to find the library function that will do what you want than it would take to write it yourself, then all that code is doing nothing but make your manual thick. (The Symbolics manuals were a case in point.) So I think we will have to work on ways to organize libraries. The ideal would be to design them so that the programmer could guess what library call would do the right thing.

2. Are People Really Scared of Prefix Syntax?

This is an open problem in the sense that I have wondered about it for years and still don't know the answer. Prefix syntax seems perfectly natural to me, except possibly for math. But it could be that a lot of Lisp's unpopularity is simply due to having an unfamiliar syntax. Whether to do anything about it, if it is true, is another question.

3. What Do You Need for Server-Based Software?

I think a lot of the most exciting new applications that get written in the next twenty years will be Web-based applications, meaning programs that sit on the server and talk to you through a Web browser. And to write these kinds of programs we may need some new things.

One thing we'll need is support for the new way that server-based apps get released. Instead of having one or two big releases a year, like desktop software, server-based apps get released as a series of small changes. You may have as many as five or ten releases a day. And as a rule everyone will always use the latest version.

You know how you can design programs to be debuggable? Well, server-based software likewise has to be designed to be changeable. You have to be able to change it easily, or at least to know what is a small change and what is a momentous one.

Another thing that might turn out to be useful for server based software, surprisingly, is continuations. In Web-based software you can use something like continuation-passing style to get the effect of [subroutines](#) in the inherently stateless world of a Web session. Maybe it would be worthwhile having actual continuations, if it was not too expensive.

4. What New Abstractions Are Left to Discover?

I'm not sure how reasonable a hope this is, but one thing I would really love to do, personally, is discover a new abstraction-- something that would make as much of a difference as having first class functions or recursion or even keyword parameters. This may be an impossible dream. These things don't get discovered that often. But I am always looking.

LITTLE–KNOWN SECRETS

1. You Can Use Whatever Language You Want.

Writing application programs used to mean writing desktop software. And in desktop software there is a big bias toward writing the application in the same language as the operating system. And so ten years ago, writing software pretty much meant writing software in C. Eventually a tradition evolved: application programs must not be written in unusual languages. And this tradition had so long to develop that nontechnical people like managers and venture capitalists also learned it.

Server-based software blows away this whole model. With server-based software you can use any language you want. Almost nobody understands this yet (especially not managers and venture capitalists). A few hackers understand it, and that's why we even hear about new, indy languages like Perl and Python. We're not hearing about Perl and Python because people are using them to write Windows apps.

What this means for us, as people interested in designing programming languages, is that there is now potentially an actual audience for our work.

2. Speed Comes from Profilers.

Language designers, or at least language implementors, like to write compilers that generate fast code. But I don't think this is what makes languages fast for users. Knuth pointed out long ago that speed only matters in a few critical bottlenecks. And anyone who's tried it knows that you can't guess where these bottlenecks are. Profilers are the answer.

Language designers are solving the wrong problem. Users don't need benchmarks to run fast. What they need is a language that can show them what parts of their own programs need to be rewritten. That's where speed comes from in practice. So maybe it would be a net win if language implementors took half the time they would have spent doing compiler optimizations and spent it writing a good profiler instead.

3. You Need an Application to Drive the Design of a Language.

This may not be an absolute rule, but it seems like the best languages all evolved together with some application they were being used to write. C was written by people who needed it for systems programming. Lisp was developed partly to do symbolic differentiation, and McCarthy was so eager to get started that he was writing differentiation programs even in the first paper on Lisp, in 1960.

It's especially good if your application solves some new problem. That will tend to drive your language to have new features that programmers need. I personally am interested in writing a language that will be good for writing server-based applications.

[During the panel, Guy Steele also made this point, with the additional suggestion that the application should not consist of writing the compiler for your language, unless your language happens to be intended for writing compilers.]

4. A Language Has to Be Good for Writing Throwaway Programs.

You know what a throwaway program is: something you write quickly for some limited task. I think if you looked around you'd find that a lot of big, serious programs started as throwaway programs. I would not be surprised if *most* programs started as throwaway programs. And so if you want to make a language that's good for writing software in general, it has to be good for writing throwaway programs, because that is the larval stage of most software.

5. Syntax Is Connected to Semantics.

It's traditional to think of syntax and semantics as being completely separate. This will sound shocking, but it may be that they aren't. I think that what you want in your language may be related to how you express it.

I was talking recently to Robert Morris, and he pointed out that operator overloading is a bigger win in languages with infix syntax. In a language with prefix syntax, any function you define is effectively an operator. If you want to define a plus for a new type of number you've made up, you can just define a new function to add them. If you do that in a language with infix syntax, there's a big difference in appearance between the use of an overloaded operator and a function call.

Ideas Whose Time Has Returned

1. New Programming Languages.

Back in the 1970s it was fashionable to design new programming languages. Recently it hasn't been. But I think server-based software will make new languages fashionable again. With server-based software, you can use any language you want, so if someone does design a language that actually seems better than others that are available, there will be people who take a risk and use it.

2. Time-Sharing.

Richard Kelsey gave this as an idea whose time has come again in the last panel, and I completely agree with him. My guess (and Microsoft's guess, it seems) is that much computing will move from the desktop onto remote servers. In other words, time-sharing is back. And I think there will need to be support for it at the language level. For example, I know that Richard and Jonathan Rees have done a lot of work implementing process scheduling within Scheme 48.

3. Efficiency.

Recently it was starting to seem that computers were finally fast enough. More and more we were starting to hear about byte code, which implies to me at least that we feel we have cycles to spare. But I don't think we will, with server-based software. Someone is going to have to pay for the servers that the software runs on, and the number of users they can support per machine will be the divisor of their capital cost.

So I think efficiency will matter, at least in computational bottlenecks. It will be especially important to do i/o fast, because server-based applications do a lot of i/o.

It may turn out that byte code is not a win, in the end. Sun and Microsoft seem to be facing off in a kind of a battle of the byte codes at the moment. But they're doing it because byte code is a convenient place to insert themselves into the process, not because byte code is in itself a good idea. It may turn out that this whole battleground gets bypassed. That would be kind of amusing.

Pitfalls and Gotchas

1. Clients.

This is just a guess, but my guess is that the winning model for most applications will be purely server-based. Designing software that works on the assumption that

everyone will have your client is like designing a society on the assumption that everyone will just be honest. It would certainly be convenient, but you have to assume it will never happen.

I think there will be a proliferation of devices that have some kind of Web access, and all you'll be able to assume about them is that they can support simple html and forms. Will you have a browser on your cell phone? Will there be a phone in your palm pilot? Will your blackberry get a bigger screen? Will you be able to browse the Web on your gameboy? Your watch? I don't know. And I don't have to know if I bet on everything just being on the server. It's just so much more robust to have all the [brains on the server](#).

2. Object-Oriented Programming.

I realize this is a controversial one, but I don't think object-oriented programming is such a big deal. I think it is a fine model for certain kinds of applications that need that specific kind of data structure, like window systems, simulations, and cad programs. But I don't see why it ought to be the model for all programming.

I think part of the reason people in big companies like object-oriented programming is because it yields a lot of what looks like work. Something that might naturally be represented as, say, a list of integers, can now be represented as a class with all kinds of scaffolding and hustle and bustle.

Another attraction of object-oriented programming is that methods give you some of the effect of first class functions. But this is old news to Lisp programmers. When you have actual first class functions, you can just use them in whatever way is appropriate to the task at hand, instead of forcing everything into a mold of classes and methods.

What this means for language design, I think, is that you shouldn't build object-oriented programming in too deeply. Maybe the answer is to offer more general, underlying stuff, and let people design whatever object systems they want as libraries.

3. Design by Committee.

Having your language designed by a committee is a big pitfall, and not just for the reasons everyone knows about. Everyone knows that committees tend to yield lumpy, inconsistent designs. But I think a greater danger is that they won't take risks. When one person is in charge he can take risks that a committee would never agree on.

Is it necessary to take risks to design a good language though? Many people might suspect that language design is something where you should stick fairly close to the conventional wisdom. I bet this isn't true. In everything else people do, reward is proportionate to risk. Why should language design be any different?

The Roots of Lisp

May 2001

(I wrote this article to help myself understand exactly what McCarthy discovered. You don't need to know this stuff to program in Lisp, but it should be helpful to anyone who wants to understand the essence of Lisp � both in the sense of its origins and its semantic core. The fact that it has such a core is one of Lisp's distinguishing features, and the reason why, unlike other languages, Lisp has dialects.)

In 1960, [John McCarthy](#) published a remarkable paper in which he did for programming something like what Euclid did for geometry. He showed how, given a handful of simple operators and a notation for functions, you can build a whole programming language. He called this language Lisp, for "List Processing," because one of his key ideas was to use a simple data structure called a *list* for both code and data.

It's worth understanding what McCarthy discovered, not just as a landmark in the history of computers, but as a model for what programming is tending to become in our own time. It seems to me that there have been two really clean, consistent models of programming so far: the C model and the Lisp model. These two seem points of high ground, with swampy lowlands between them. As computers have grown more powerful, the new languages being developed have been [moving steadily](#) toward the Lisp model. A popular recipe for new programming languages in the past 20 years has been to take the C model of computing and add to it, piecemeal, parts taken from the Lisp model, like runtime typing and garbage collection.

In this article I'm going to try to explain in the simplest possible terms what McCarthy discovered. The point is not just to learn about an interesting theoretical result someone figured out forty years ago, but to show where languages are heading. The unusual thing about Lisp � in fact, the defining quality of Lisp � is that it can be written in itself. To understand what McCarthy meant by this, we're going to retrace his steps, with his mathematical notation translated into running Common Lisp code.

The Other Road Ahead

September 2001

(This article explains why much of the next generation of software may be server-based, what that will mean for programmers, and why this new kind of software is a great opportunity for startups. It's derived from a talk at BBN Labs.)

In the summer of 1995, my friend Robert Morris and I decided to start a startup. The PR campaign leading up to Netscape's IPO was running full blast then, and there was a lot of talk in the press about online commerce. At the time there might have been thirty actual stores on the Web, all made by hand. If there were going to be a lot of online stores, there would need to be software for making them, so we decided to write some.

For the first week or so we intended to make this an ordinary desktop application. Then one day we had the idea of making the software run on our Web server, using the browser as an interface. We tried rewriting the software to work over the Web, and it was clear that this was the way to go. If we wrote our software to run on the server, it would be a lot easier for the users and for us as well.

This turned out to be a good plan. Now, as [Yahoo Store](), this software is the most popular online store builder, with about 14,000 users.

When we started Viaweb, hardly anyone understood what we meant when we said that the software ran on the server. It was not until Hotmail was launched a year later that people started to get it. Now everyone knows that this is a valid approach. There is a name now for what we were: an Application Service Provider, or ASP.

I think that a lot of the next generation of software will be written on this model. Even Microsoft, who have the most to lose, seem to see the inevitablity of moving some things off the desktop. If software moves off the desktop and onto servers, it will mean a very different world for developers. This article describes the surprising things we saw, as some of the first visitors to this new world. To the extent software does move onto servers, what I'm describing here is the future.

The Next Thing?

When we look back on the desktop software era, I think we'll marvel at the

inconveniences people put up with, just as we marvel now at what early car owners put up with. For the first twenty or thirty years, you had to be a car expert to own a car. But cars were such a big win that lots of people who weren't car experts wanted to have them as well.

Computers are in this phase now. When you own a desktop computer, you end up learning a lot more than you wanted to know about what's happening inside it. But more than half the households in the US own one. My mother has a computer that she uses for email and for keeping accounts. About a year ago she was alarmed to receive a letter from Apple, offering her a discount on a new version of the operating system. There's something wrong when a sixty-five year old woman who wants to use a computer for email and accounts has to think about installing new operating systems. Ordinary users shouldn't even know the words "operating system," much less "device driver" or "patch."

There is now another way to deliver software that will save users from becoming system administrators. Web-based applications are programs that run on Web servers and use Web pages as the user interface. For the average user this new kind of software will be easier, cheaper, more mobile, more reliable, and often more powerful than desktop software.

With Web-based software, most users won't have to think about anything except the applications they use. All the messy, changing stuff will be sitting on a server somewhere, maintained by the kind of people who are good at that kind of thing. And so you won't ordinarily need a computer, per se, to use software. All you'll need will be something with a keyboard, a screen, and a Web browser. Maybe it will have wireless Internet access. Maybe it will also be your cell phone. Whatever it is, it will be consumer electronics: something that costs about $200, and that people choose mostly based on how the case looks. You'll pay more for Internet services than you do for the hardware, just as you do now with telephones. [1]

It will take about a tenth of a second for a click to get to the server and back, so users of heavily interactive software, like Photoshop, will still want to have the computations happening on the desktop. But if you look at the kind of things most people use computers for, a tenth of a second latency would not be a problem. My mother doesn't really need a desktop computer, and there are a lot of people like her.

The Win for Users

Near my house there is a car with a bumper sticker that reads "death before inconvenience." Most people, most of the time, will take whatever choice requires least work. If Web-based software wins, it will be because it's more convenient. And it looks as if it will be, for users and developers both.

To use a purely Web-based application, all you need is a browser connected to the Internet. So you can use a Web-based application anywhere. When you install software on your desktop computer, you can only use it on that computer. Worse

still, your files are trapped on that computer. The inconvenience of this model becomes more and more evident as people get used to networks.

The thin end of the wedge here was Web-based email. Millions of people now realize that you should have access to email messages no matter where you are. And if you can see your email, why not your calendar? If you can discuss a document with your colleagues, why can't you edit it? Why should any of your data be trapped on some computer sitting on a faraway desk?

The whole idea of "your computer" is going away, and being replaced with "your data." You should be able to get at your data from any computer. Or rather, any client, and a client doesn't have to be a computer.

Clients shouldn't store data; they should be like telephones. In fact they may become telephones, or vice versa. And as clients get smaller, you have another reason not to keep your data on them: something you carry around with you can be lost or stolen. Leaving your PDA in a taxi is like a disk crash, except that your data is handed to someone else instead of being vaporized.

With purely Web-based software, neither your data nor the applications are kept on the client. So you don't have to install anything to use it. And when there's no installation, you don't have to worry about installation going wrong. There can't be incompatibilities between the application and your operating system, because the software doesn't run on your operating system.

Because it needs no installation, it will be easy, and common, to try Web-based software before you "buy" it. You should expect to be able to test-drive any Web-based application for free, just by going to the site where it's offered. At Viaweb our whole site was like a big arrow pointing users to the test drive.

After trying the demo, signing up for the service should require nothing more than filling out a brief form (the briefer the better). And that should be the last work the user has to do. With Web-based software, you should get new releases without paying extra, or doing any work, or possibly even knowing about it.

Upgrades won't be the big shocks they are now. Over time applications will quietly grow more powerful. This will take some effort on the part of the developers. They will have to design software so that it can be updated without confusing the users. That's a new problem, but there are ways to solve it.

With Web-based applications, everyone uses the same version, and bugs can be fixed as soon as they're discovered. So Web-based software should have far fewer bugs than desktop software. At Viaweb, I doubt we ever had ten known bugs at any one time. That's orders of magnitude better than desktop software.

Web-based applications can be used by several people at the same time. This is an obvious win for collaborative applications, but I bet users will start to want this in most applications once they realize it's possible. It will often be useful to let two

people edit the same document, for example. Viaweb let multiple users edit a site simultaneously, more because that was the right way to write the software than because we expected users to want to, but it turned out that many did.

When you use a Web-based application, your data will be safer. Disk crashes won't be a thing of the past, but users won't hear about them anymore. They'll happen within server farms. And companies offering Web-based applications will actually do backups-- not only because they'll have real system administrators worrying about such things, but because an ASP that does lose people's data will be in big, big trouble. When people lose their own data in a disk crash, they can't get that mad, because they only have themselves to be mad at. When a company loses their data for them, they'll get a lot madder.

Finally, Web-based software should be less vulnerable to viruses. If the client doesn't run anything except a browser, there's less chance of running viruses, and no data locally to damage. And a program that attacked the servers themselves should find them very well defended. [2]

For users, Web-based software will be *less stressful.* I think if you looked inside the average Windows user you'd find a huge and pretty much untapped desire for software meeting that description. Unleashed, it could be a powerful force.

City of Code

To developers, the most conspicuous difference between Web-based and desktop software is that a Web-based application is not a single piece of code. It will be a collection of programs of different types rather than a single big binary. And so designing Web-based software is like desiging a city rather than a building: as well as buildings you need roads, street signs, utilities, police and fire departments, and plans for both growth and various kinds of disasters.

At Viaweb, software included fairly big applications that users talked to directly, programs that those programs used, programs that ran constantly in the background looking for problems, programs that tried to restart things if they broke, programs that ran occasionally to compile statistics or build indexes for searches, programs we ran explicitly to garbage-collect resources or to move or restore data, programs that pretended to be users (to measure performance or expose bugs), programs for diagnosing network troubles, programs for doing backups, interfaces to outside services, software that drove an impressive collection of dials displaying real-time server statistics (a hit with visitors, but indispensable for us too), modifications (including bug fixes) to open-source software, and a great many configuration files and settings. Trevor Blackwell wrote a spectacular program for moving stores to new servers across the country, without shutting them down, after we were bought by Yahoo. Programs paged us, sent faxes and email to users, conducted transactions with credit card processors, and talked to one another through sockets, pipes, http requests, ssh, udp packets, shared memory, and files. Some of Viaweb even consisted of the absence of programs, since one of the keys to Unix security is not to run unnecessary utilities

that people might use to break into your servers.

It did not end with software. We spent a lot of time thinking about server configurations. We built the servers ourselves, from components-- partly to save money, and partly to get exactly what we wanted. We had to think about whether our upstream ISP had fast enough connections to all the backbones. We serially dated RAID suppliers.

But hardware is not just something to worry about. When you control it you can do more for users. With a desktop application, you can specify certain minimum hardware, but you can't add more. If you administer the servers, you can in one step enable all your users to page people, or send faxes, or send commands by phone, or process credit cards, etc, just by installing the relevant hardware. We always looked for new ways to add features with hardware, not just because it pleased users, but also as a way to distinguish ourselves from competitors who (either because they sold desktop software, or resold Web-based applications through ISPs) didn't have direct control over the hardware.

Because the software in a Web-based application will be a collection of programs rather than a single binary, it can be written in any number of different languages. When you're writing desktop software, you're practically forced to write the application in the same language as the underlying operating system-- meaning C and C++. And so these languages (especially among nontechnical people like managers and VCs) got to be considered as the languages for "serious" software development. But that was just an artifact of the way desktop software had to be delivered. For server-based software you can use any language you want. [3] Today a lot of the top hackers are using languages far removed from C and C++: Perl, Python, and even Lisp.

With server-based software, no one can tell you what language to use, because you control the whole system, right down to the hardware. Different languages are good for different tasks. You can use whichever is best for each. And when you have competitors, "you can" means "you must" (we'll return to this later), because if you don't take advantage of this possibility, your competitors will.

Most of our competitors used C and C++, and this made their software visibly inferior because (among other things), they had no way around the statelessness of CGI scripts. If you were going to change something, all the changes had to happen on one page, with an Update button at the bottom. As I've written elsewhere, by using Lisp, which many people still consider a research language, we could make the Viaweb editor behave more like desktop software.

Releases

One of the most important changes in this new world is the way you do releases. In the desktop software business, doing a release is a huge trauma, in which the whole company sweats and strains to push out a single, giant piece of code. Obvious comparisons suggest themselves, both to the process and the resulting

product.

With server-based software, you can make changes almost as you would in a program you were writing for yourself. You release software as a series of incremental changes instead of an occasional big explosion. A typical desktop software company might do one or two releases a year. At Viaweb we often did three to five releases a day.

When you switch to this new model, you realize how much software development is affected by the way it is released. Many of the nastiest problems you see in the desktop software business are due to catastrophic nature of releases.

When you release only one new version a year, you tend to deal with bugs wholesale. Some time before the release date you assemble a new version in which half the code has been torn out and replaced, introducing countless bugs. Then a squad of QA people step in and start counting them, and the programmers work down the list, fixing them. They do not generally get to the end of the list, and indeed, no one is sure where the end is. It's like fishing rubble out of a pond. You never really know what's happening inside the software. At best you end up with a statistical sort of correctness.

With server-based software, most of the change is small and incremental. That in itself is less likely to introduce bugs. It also means you know what to test most carefully when you're about to release software: the last thing you changed. You end up with a much firmer grip on the code. As a general rule, you do know what's happening inside it. You don't have the source code memorized, of course, but when you read the source you do it like a pilot scanning the instrument panel, not like a detective trying to unravel some mystery.

Desktop software breeds a certain fatalism about bugs. You know that you're shipping something loaded with bugs, and you've even set up mechanisms to compensate for it (e.g. patch releases). So why worry about a few more? Soon you're releasing whole features you know are broken. Apple did this earlier this year. They felt under pressure to release their new OS, whose release date had already slipped four times, but some of the software (support for CDs and DVDs) wasn't ready. The solution? They released the OS without the unfinished parts, and users will have to install them later.

With Web-based software, you never have to release software before it works, and you can release it as soon as it does work.

The industry veteran may be thinking, it's a fine-sounding idea to say that you never have to release software before it works, but what happens when you've promised to deliver a new version of your software by a certain date? With Web-based software, you wouldn't make such a promise, because there are no versions. Your software changes gradually and continuously. Some changes might be bigger than others, but the idea of versions just doesn't naturally fit onto Web-based software.

If anyone remembers Viaweb this might sound odd, because we were always announcing new versions. This was done entirely for PR purposes. The trade press, we learned, thinks in version numbers. They will give you major coverage for a major release, meaning a new first digit on the version number, and generally a paragraph at most for a point release, meaning a new digit after the decimal point.

Some of our competitors were offering desktop software and actually had version numbers. And for these releases, the mere fact of which seemed to us evidence of their backwardness, they would get all kinds of publicity. We didn't want to miss out, so we started giving version numbers to our software too. When we wanted some publicity, we'd make a list of all the features we'd added since the last "release," stick a new version number on the software, and issue a press release saying that the new version was available immediately. Amazingly, no one ever called us on it.

By the time we were bought, we had done this three times, so we were on Version 4. Version 4.1 if I remember correctly. After Viaweb became Yahoo Store, there was no longer such a desperate need for publicity, so although the software continued to evolve, the whole idea of version numbers was quietly dropped.

Bugs

The other major technical advantage of Web-based software is that you can reproduce most bugs. You have the users' data right there on your disk. If someone breaks your software, you don't have to try to guess what's going on, as you would with desktop software: you should be able to reproduce the error while they're on the phone with you. You might even know about it already, if you have code for noticing errors built into your application.

Web-based software gets used round the clock, so everything you do is immediately put through the wringer. Bugs turn up quickly.

Software companies are sometimes accused of letting the users debug their software. And that is just what I'm advocating. For Web-based software it's actually a good plan, because the bugs are fewer and transient. When you release software gradually you get far fewer bugs to start with. And when you can reproduce errors and release changes instantly, you can find and fix most bugs as soon as they appear. We never had enough bugs at any one time to bother with a formal bug-tracking system.

You should test changes before you release them, of course, so no major bugs should get released. Those few that inevitably slip through will involve borderline cases and will only affect the few users that encounter them before someone calls in to complain. As long as you fix bugs right away, the net effect, for the average user, is far fewer bugs. I doubt the average Viaweb user ever saw a bug.

Fixing fresh bugs is easier than fixing old ones. It's usually fairly quick to find a

bug in code you just wrote. When it turns up you often know what's wrong before you even look at the source, because you were already worrying about it subconsciously. Fixing a bug in something you wrote six months ago (the average case if you release once a year) is a lot more work. And since you don't understand the code as well, you're more likely to fix it in an ugly way, or even introduce more bugs. [4]

When you catch bugs early, you also get fewer compound bugs. Compound bugs are two separate bugs that interact: you trip going downstairs, and when you reach for the handrail it comes off in your hand. In software this kind of bug is the hardest to find, and also tends to have the worst consequences. [5] The traditional "break everything and then filter out the bugs" approach inherently yields a lot of compound bugs. And software that's released in a series of small changes inherently tends not to. The floors are constantly being swept clean of any loose objects that might later get stuck in something.

It helps if you use a technique called functional programming. Functional programming means avoiding side-effects. It's something you're more likely to see in research papers than commercial software, but for Web-based applications it turns out to be really useful. It's hard to write entire programs as purely functional code, but you can write substantial chunks this way. It makes those parts of your software easier to test, because they have no state, and that is very convenient in a situation where you are constantly making and testing small modifications. I wrote much of Viaweb's editor in this style, and we made our scripting language, RTML, a purely functional language.

People from the desktop software business will find this hard to credit, but at Viaweb bugs became almost a game. Since most released bugs involved borderline cases, the users who encountered them were likely to be advanced users, pushing the envelope. Advanced users are more forgiving about bugs, especially since you probably introduced them in the course of adding some feature they were asking for. In fact, because bugs were rare and you had to be doing sophisticated things to see them, advanced users were often proud to catch one. They would call support in a spirit more of triumph than anger, as if they had scored points off us.

Support

When you can reproduce errors, it changes your approach to customer support. At most software companies, support is offered as a way to make customers feel better. They're either calling you about a known bug, or they're just doing something wrong and you have to figure out what. In either case there's not much you can learn from them. And so you tend to view support calls as a pain in the ass that you want to isolate from your developers as much as possible.

This was not how things worked at Viaweb. At Viaweb, support was free, because we wanted to hear from customers. If someone had a problem, we wanted to know about it right away so that we could reproduce the error and release a fix.

So at Viaweb the developers were always in close contact with support. The customer support people were about thirty feet away from the programmers, and knew that they could always interrupt anything with a report of a genuine bug. We would leave a board meeting to fix a serious bug.

Our approach to support made everyone happier. The customers were delighted. Just imagine how it would feel to call a support line and be treated as someone bringing important news. The customer support people liked it because it meant they could help the users, instead of reading scripts to them. And the programmers liked it because they could reproduce bugs instead of just hearing vague second-hand reports about them.

Our policy of fixing bugs on the fly changed the relationship between customer support people and hackers. At most software companies, support people are underpaid human shields, and hackers are little copies of God the Father, creators of the world. Whatever the procedure for reporting bugs, it is likely to be one-directional: support people who hear about bugs fill out some form that eventually gets passed on (possibly via QA) to programmers, who put it on their list of things to do. It was very different at Viaweb. Within a minute of hearing about a bug from a customer, the support people could be standing next to a programmer hearing him say "Shit, you're right, it's a bug." It delighted the support people to hear that "you're right" from the hackers. They used to bring us bugs with the same expectant air as a cat bringing you a mouse it has just killed. It also made them more careful in judging the seriousness of a bug, because now their honor was on the line.

After we were bought by Yahoo, the customer support people were moved far away from the programmers. It was only then that we realized that they were effectively QA and to some extent marketing as well. In addition to catching bugs, they were the keepers of the knowledge of vaguer, buglike things, like features that confused users. [6] They were also a kind of proxy focus group; we could ask them which of two new features users wanted more, and they were always right.

Morale

Being able to release software immediately is a big motivator. Often as I was walking to work I would think of some change I wanted to make to the software, and do it that day. This worked for bigger features as well. Even if something was going to take two weeks to write (few projects took longer), I knew I could see the effect in the software as soon as it was done.

If I'd had to wait a year for the next release, I would have shelved most of these ideas, for a while at least. The thing about ideas, though, is that they lead to more ideas. Have you ever noticed that when you sit down to write something, half the ideas that end up in it are ones you thought of while writing it? The same thing happens with software. Working to implement one idea gives you more ideas. So shelving an idea costs you not only that delay in implementing it, but also all the ideas that implementing it would have led to. In fact, shelving an idea probably

even inhibits new ideas: as you start to think of some new feature, you catch sight of the shelf and think "but I already have a lot of new things I want to do for the next release."

What big companies do instead of implementing features is plan them. At Viaweb we sometimes ran into trouble on this account. Investors and analysts would ask us what we had planned for the future. The truthful answer would have been, we didn't have any plans. We had general ideas about things we wanted to improve, but if we knew how we would have done it already. What were we going to do in the next six months? Whatever looked like the biggest win. I don't know if I ever dared give this answer, but that was the truth. Plans are just another word for ideas on the shelf. When we thought of good ideas, we implemented them.

At Viaweb, as at many software companies, most code had one definite owner. But when you owned something you really owned it: no one except the owner of a piece of software had to approve (or even know about) a release. There was no protection against breakage except the fear of looking like an idiot to one's peers, and that was more than enough. I may have given the impression that we just blithely plowed forward writing code. We did go fast, but we thought very carefully before we released software onto those servers. And paying attention is more important to reliability than moving slowly. Because he pays close attention, a Navy pilot can land a 40,000 lb. aircraft at 140 miles per hour on a pitching carrier deck, at night, more safely than the average teenager can cut a bagel.

This way of writing software is a double-edged sword of course. It works a lot better for a small team of good, trusted programmers than it would for a big company of mediocre ones, where bad ideas are caught by committees instead of the people that had them.

Brooks in Reverse

Fortunately, Web-based software does require fewer programmers. I once worked for a medium-sized desktop software company that had over 100 people working in engineering as a whole. Only 13 of these were in product development. All the rest were working on releases, ports, and so on. With Web-based software, all you need (at most) are the 13 people, because there are no releases, ports, and so on.

Viaweb was written by just three people. [7] I was always under pressure to hire more, because we wanted to get bought, and we knew that buyers would have a hard time paying a high price for a company with only three programmers. (Solution: we hired more, but created new projects for them.)

When you can write software with fewer programmers, it saves you more than money. As Fred Brooks pointed out in *The Mythical Man-Month,* adding people to a project tends to slow it down. The number of possible connections between developers grows exponentially with the size of the group. The larger the group, the more time they'll spend in meetings negotiating how their software will work together, and the more bugs they'll get from unforeseen interactions. Fortunately,

this process also works in reverse: as groups get smaller, software development gets exponentially more efficient. I can't remember the programmers at Viaweb ever having an actual meeting. We never had more to say at any one time than we could say as we were walking to lunch.

If there is a downside here, it is that all the programmers have to be to some degree system administrators as well. When you're hosting software, someone has to be watching the servers, and in practice the only people who can do this properly are the ones who wrote the software. At Viaweb our system had so many components and changed so frequently that there was no definite border between software and infrastructure. Arbitrarily declaring such a border would have constrained our design choices. And so although we were constantly hoping that one day ("in a couple months") everything would be stable enough that we could hire someone whose job was just to worry about the servers, it never happened.

I don't think it could be any other way, as long as you're still actively developing the product. Web-based software is never going to be something you write, check in, and go home. It's a live thing, running on your servers right now. A bad bug might not just crash one user's process; it could crash them all. If a bug in your code corrupts some data on disk, you have to fix it. And so on. We found that you don't have to watch the servers every minute (after the first year or so), but you definitely want to keep an eye on things you've changed recently. You don't release code late at night and then go home.

Watching Users

With server-based software, you're in closer touch with your code. You can also be in closer touch with your users. Intuit is famous for introducing themselves to customers at retail stores and asking to follow them home. If you've ever watched someone use your software for the first time, you know what surprises must have awaited them.

Software should do what users think it will. But you can't have any idea what users will be thinking, believe me, until you watch them. And server-based software gives you unprecedented information about their behavior. You're not limited to small, artificial focus groups. You can see every click made by every user. You have to consider carefully what you're going to look at, because you don't want to violate users' privacy, but even the most general statistical sampling can be very useful.

When you have the users on your server, you don't have to rely on benchmarks, for example. Benchmarks are simulated users. With server-based software, you can watch actual users. To decide what to optimize, just log into a server and see what's consuming all the CPU. And you know when to stop optimizing too: we eventually got the Viaweb editor to the point where it was memory-bound rather than CPU-bound, and since there was nothing we could do to decrease the size of users' data (well, nothing easy), we knew we might as well stop there.

Efficiency matters for server-based software, because you're paying for the hardware. The number of users you can support per server is the divisor of your capital cost, so if you can make your software very efficient you can undersell competitors and still make a profit. At Viaweb we got the capital cost per user down to about $5. It would be less now, probably less than the cost of sending them the first month's bill. Hardware is free now, if your software is reasonably efficient.

Watching users can guide you in design as well as optimization. Viaweb had a scripting language called RTML that let advanced users define their own page styles. We found that RTML became a kind of suggestion box, because users only used it when the predefined page styles couldn't do what they wanted. Originally the editor put button bars across the page, for example, but after a number of users used RTML to put buttons down the left side, we made that an option (in fact the default) in the predefined page styles.

Finally, by watching users you can often tell when they're in trouble. And since the customer is always right, that's a sign of something you need to fix. At Viaweb the key to getting users was the online test drive. It was not just a series of slides built by marketing people. In our test drive, users actually used the software. It took about five minutes, and at the end of it they had built a real, working store.

The test drive was the way we got nearly all our new users. I think it will be the same for most Web-based applications. If users can get through a test drive successfully, they'll like the product. If they get confused or bored, they won't. So anything we could do to get more people through the test drive would increase our growth rate.

I studied click trails of people taking the test drive and found that at a certain step they would get confused and click on the browser's Back button. (If you try writing Web-based applications, you'll find that the Back button becomes one of your most interesting philosophical problems.) So I added a message at that point, telling users that they were nearly finished, and reminding them not to click on the Back button. Another great thing about Web-based software is that you get instant feedback from changes: the number of people completing the test drive rose immediately from 60% to 90%. And since the number of new users was a function of the number of completed test drives, our revenue growth increased by 50%, just from that change.

Money

In the early 1990s I read an article in which someone said that software was a subscription business. At first this seemed a very cynical statement. But later I realized that it reflects reality: software development is an ongoing process. I think it's cleaner if you openly charge subscription fees, instead of forcing people to keep buying and installing new versions so that they'll keep paying you. And fortunately, subscriptions are the natural way to bill for Web-based applications.

Hosting applications is an area where companies will play a role that is not likely to be filled by freeware. Hosting applications is a lot of stress, and has real expenses. No one is going to want to do it for free.

For companies, Web-based applications are an ideal source of revenue. Instead of starting each quarter with a blank slate, you have a recurring revenue stream. Because your software evolves gradually, you don't have to worry that a new model will flop; there never need be a new model, per se, and if you do something to the software that users hate, you'll know right away. You have no trouble with uncollectable bills; if someone won't pay you can just turn off the service. And there is no possibility of piracy.

That last "advantage" may turn out to be a problem. Some amount of piracy is to the advantage of software companies. If some user really would not have bought your software at any price, you haven't lost anything if he uses a pirated copy. In fact you gain, because he is one more user helping to make your software the standard-- or who might buy a copy later, when he graduates from high school.

When they can, companies like to do something called price discrimination, which means charging each customer as much as they can afford. [8] Software is particularly suitable for price discrimination, because the marginal cost is close to zero. This is why some software costs more to run on Suns than on Intel boxes: a company that uses Suns is not interested in saving money and can safely be charged more. Piracy is effectively the lowest tier of price discrimination. I think that software companies understand this and deliberately turn a blind eye to some kinds of piracy. [9] With server-based software they are going to have to come up with some other solution.

Web-based software sells well, especially in comparison to desktop software, because it's easy to buy. You might think that people decide to buy something, and then buy it, as two separate steps. That's what I thought before Viaweb, to the extent I thought about the question at all. In fact the second step can propagate back into the first: if something is hard to buy, people will change their mind about whether they wanted it. And vice versa: you'll sell more of something when it's easy to buy. I buy more books because Amazon exists. Web-based software is just about the easiest thing in the world to buy, especially if you have just done an online demo. Users should not have to do much more than enter a credit card number. (Make them do more at your peril.)

Sometimes Web-based software is offered through ISPs acting as resellers. This is a bad idea. You have to be administering the servers, because you need to be constantly improving both hardware and software. If you give up direct control of the servers, you give up most of the advantages of developing Web-based applications.

Several of our competitors shot themselves in the foot this way-- usually, I think, because they were overrun by suits who were excited about this huge potential channel, and didn't realize that it would ruin the product they hoped to sell through

it. Selling Web-based software through ISPs is like selling sushi through vending machines.

Customers

Who will the customers be? At Viaweb they were initially individuals and smaller companies, and I think this will be the rule with Web-based applications. These are the users who are ready to try new things, partly because they're more flexible, and partly because they want the lower costs of new technology.

Web-based applications will often be the best thing for big companies too (though they'll be slow to realize it). The best intranet is the Internet. If a company uses true Web-based applications, the software will work better, the servers will be better administered, and employees will have access to the system from anywhere.

The argument against this approach usually hinges on security: if access is easier for employees, it will be for bad guys too. Some larger merchants were reluctant to use Viaweb because they thought customers' credit card information would be safer on their own servers. It was not easy to make this point diplomatically, but in fact the data was almost certainly safer in our hands than theirs. Who can hire better people to manage security, a technology startup whose whole business is running servers, or a clothing retailer? Not only did we have better people worrying about security, we worried more about it. If someone broke into the clothing retailer's servers, it would affect at most one merchant, could probably be hushed up, and in the worst case might get one person fired. If someone broke into ours, it could affect thousands of merchants, would probably end up as news on CNet, and could put us out of business.

If you want to keep your money safe, do you keep it under your mattress at home, or put it in a bank? This argument applies to every aspect of server administration: not just security, but uptime, bandwidth, load management, backups, etc. Our existence depended on doing these things right. Server problems were the big no-no for us, like a dangerous toy would be for a toy maker, or a salmonella outbreak for a food processor.

A big company that uses Web-based applications is to that extent outsourcing IT. Drastic as it sounds, I think this is generally a good idea. Companies are likely to get better service this way than they would from in-house system administrators. System administrators can become cranky and unresponsive because they're not directly exposed to competitive pressure: a salesman has to deal with customers, and a developer has to deal with competitors' software, but a system administrator, like an old bachelor, has few external forces to keep him in line. [10] At Viaweb we had external forces in plenty to keep us in line. The people calling us were customers, not just co-workers. If a server got wedged, we jumped; just thinking about it gives me a jolt of adrenaline, years later.

So Web-based applications will ordinarily be the right answer for big companies

too. They will be the last to realize it, however, just as they were with desktop computers. And partly for the same reason: it will be worth a lot of money to convince big companies that they need something more expensive.

There is always a tendency for rich customers to buy expensive solutions, even when cheap solutions are better, because the people offering expensive solutions can spend more to sell them. At Viaweb we were always up against this. We lost several high-end merchants to Web consulting firms who convinced them they'd be better off if they paid half a million dollars for a custom-made online store on their own server. They were, as a rule, not better off, as more than one discovered when Christmas shopping season came around and loads rose on their server. Viaweb was a lot more sophisticated than what most of these merchants got, but we couldn't afford to tell them. At $300 a month, we couldn't afford to send a team of well-dressed and authoritative-sounding people to make presentations to customers.

A large part of what big companies pay extra for is the cost of selling expensive things to them. (If the Defense Department pays a thousand dollars for toilet seats, it's partly because it costs a lot to sell toilet seats for a thousand dollars.) And this is one reason intranet software will continue to thrive, even though it is probably a bad idea. It's simply more expensive. There is nothing you can do about this conundrum, so the best plan is to go for the smaller customers first. The rest will come in time.

Son of Server

Running software on the server is nothing new. In fact it's the old model: mainframe applications are all server-based. If server-based software is such a good idea, why did it lose last time? Why did desktop computers eclipse mainframes?

At first desktop computers didn't look like much of a threat. The first users were all hackers-- or hobbyists, as they were called then. They liked microcomputers because they were cheap. For the first time, you could have your own computer. The phrase "personal computer" is part of the language now, but when it was first used it had a deliberately audacious sound, like the phrase "personal satellite" would today.

Why did desktop computers take over? I think it was because they had better software. And I think the reason microcomputer software was better was that it could be written by small companies.

I don't think many people realize how fragile and tentative startups are in the earliest stage. Many startups begin almost by accident-- as a couple guys, either with day jobs or in school, writing a prototype of something that might, if it looks promising, turn into a company. At this larval stage, any significant obstacle will stop the startup dead in its tracks. Writing mainframe software required too much commitment up front. Development machines were expensive, and because the

customers would be big companies, you'd need an impressive-looking sales force to sell it to them. Starting a startup to write mainframe software would be a much more serious undertaking than just hacking something together on your Apple II in the evenings. And so you didn't get a lot of startups writing mainframe applications.

The arrival of desktop computers inspired a lot of new software, because writing applications for them seemed an attainable goal to larval startups. Development was cheap, and the customers would be individual people that you could reach through computer stores or even by mail-order.

The application that pushed desktop computers out into the mainstream was VisiCalc, the first spreadsheet. It was written by two guys working in an attic, and yet did things no mainframe software could do. [11] VisiCalc was such an advance, in its time, that people bought Apple IIs just to run it. And this was the beginning of a trend: desktop computers won because startups wrote software for them.

It looks as if server-based software will be good this time around, because startups will write it. Computers are so cheap now that you can get started, as we did, using a desktop computer as a server. Inexpensive processors have eaten the workstation market (you rarely even hear the word now) and are most of the way through the server market; Yahoo's servers, which deal with loads as high as any on the Internet, all have the same inexpensive Intel processors that you have in your desktop machine. And once you've written the software, all you need to sell it is a Web site. Nearly all our users came direct to our site through word of mouth and references in the press. [12]

Viaweb was a typical larval startup. We were terrified of starting a company, and for the first few months comforted ourselves by treating the whole thing as an experiment that we might call off at any moment. Fortunately, there were few obstacles except technical ones. While we were writing the software, our Web server was the same desktop machine we used for development, connected to the outside world by a dialup line. Our only expenses in that phase were food and rent.

There is all the more reason for startups to write Web-based software now, because writing desktop software has become a lot less fun. If you want to write desktop software now you do it on Microsoft's terms, calling their APIs and working around their buggy OS. And if you manage to write something that takes off, you may find that you were merely doing market research for Microsoft.

If a company wants to make a platform that startups will build on, they have to make it something that hackers themselves will want to use. That means it has to be inexpensive and well-designed. The Mac was popular with hackers when it first came out, and a lot of them wrote software for it. [13] You see this less with Windows, because hackers don't use it. The kind of people who are good at writing software tend to be running Linux or FreeBSD now.

I don't think we would have started a startup to write desktop software, because

desktop software has to run on Windows, and before we could write software for Windows we'd have to use it. The Web let us do an end-run around Windows, and deliver software running on Unix direct to users through the browser. That is a liberating prospect, a lot like the arrival of PCs twenty-five years ago.

Microsoft

Back when desktop computers arrived, IBM was the giant that everyone was afraid of. It's hard to imagine now, but I remember the feeling very well. Now the frightening giant is Microsoft, and I don't think they are as blind to the threat facing them as IBM was. After all, Microsoft deliberately built their business in IBM's blind spot.

I mentioned earlier that my mother doesn't really need a desktop computer. Most users probably don't. That's a problem for Microsoft, and they know it. If applications run on remote servers, no one needs Windows. What will Microsoft do? Will they be able to use their control of the desktop to prevent, or constrain, this new generation of software?

My guess is that Microsoft will develop some kind of server/desktop hybrid, where the operating system works together with servers they control. At a minimum, files will be centrally available for users who want that. I don't expect Microsoft to go all the way to the extreme of doing the computations on the server, with only a browser for a client, if they can avoid it. If you only need a browser for a client, you don't need Microsoft on the client, and if Microsoft doesn't control the client, they can't push users towards their server-based applications.

I think Microsoft will have a hard time keeping the genie in the bottle. There will be too many different types of clients for them to control them all. And if Microsoft's applications only work with some clients, competitors will be able to trump them by offering applications that work from any client. [14]

In a world of Web-based applications, there is no automatic place for Microsoft. They may succeed in making themselves a place, but I don't think they'll dominate this new world as they did the world of desktop applications.

It's not so much that a competitor will trip them up as that they will trip over themselves. With the rise of Web-based software, they will be facing not just technical problems but their own wishful thinking. What they need to do is cannibalize their existing business, and I can't see them facing that. The same single-mindedness that has brought them this far will now be working against them. IBM was in exactly the same situation, and they could not master it. IBM made a late and half-hearted entry into the microcomputer business because they were ambivalent about threatening their cash cow, mainframe computing. Microsoft will likewise be hampered by wanting to save the desktop. A cash cow can be a damned heavy monkey on your back.

I'm not saying that no one will dominate server-based applications. Someone

probably will eventually. But I think that there will be a good long period of cheerful chaos, just as there was in the early days of microcomputers. That was a good time for startups. Lots of small companies flourished, and did it by making cool things.

Startups but More So

The classic startup is fast and informal, with few people and little money. Those few people work very hard, and technology magnifies the effect of the decisions they make. If they win, they win big.

In a startup writing Web-based applications, everything you associate with startups is taken to an extreme. You can write and launch a product with even fewer people and even less money. You have to be even faster, and you can get away with being more informal. You can literally launch your product as three guys sitting in the living room of an apartment, and a server collocated at an ISP. We did.

Over time the teams have gotten smaller, faster, and more informal. In 1960, software development meant a roomful of men with horn rimmed glasses and narrow black neckties, industriously writing ten lines of code a day on IBM coding forms. In 1980, it was a team of eight to ten people wearing jeans to the office and typing into vt100s. Now it's a couple of guys sitting in a living room with laptops. (And jeans turn out not to be the last word in informality.)

Startups are stressful, and this, unfortunately, is also taken to an extreme with Web-based applications. Many software companies, especially at the beginning, have periods where the developers slept under their desks and so on. The alarming thing about Web-based software is that there is nothing to prevent this becoming the default. The stories about sleeping under desks usually end: then at last we shipped it and we all went home and slept for a week. Web-based software never ships. You can work 16-hour days for as long as you want to. And because you can, and your competitors can, you tend to be forced to. You can, so you must. It's Parkinson's Law running in reverse.

The worst thing is not the hours but the responsibility. Programmers and system administrators traditionally each have their own separate worries. Programmers have to worry about bugs, and system administrators have to worry about infrastructure. Programmers may spend a long day up to their elbows in source code, but at some point they get to go home and forget about it. System administrators never quite leave the job behind, but when they do get paged at 4:00 AM, they don't usually have to do anything very complicated. With Web-based applications, these two kinds of stress get combined. The programmers become system administrators, but without the sharply defined limits that ordinarily make the job bearable.

At Viaweb we spent the first six months just writing software. We worked the usual long hours of an early startup. In a desktop software company, this would have been the part where we were working hard, but it felt like a vacation compared to

the next phase, when we took users onto our server. The second biggest benefit of selling Viaweb to Yahoo (after the money) was to be able to dump ultimate responsibility for the whole thing onto the shoulders of a big company.

Desktop software forces users to become system administrators. Web-based software forces programmers to. There is less stress in total, but more for the programmers. That's not necessarily bad news. If you're a startup competing with a big company, it's good news. [15] Web-based applications offer a straightforward way to outwork your competitors. No startup asks for more.

Just Good Enough

One thing that might deter you from writing Web-based applications is the lameness of Web pages as a UI. That is a problem, I admit. There were a few things we would have *really* liked to add to HTML and HTTP. What matters, though, is that Web pages are just good enough.

There is a parallel here with the first microcomputers. The processors in those machines weren't actually intended to be the CPUs of computers. They were designed to be used in things like traffic lights. But guys like Ed Roberts, who designed the Altair, realized that they were just good enough. You could combine one of these chips with some memory (256 bytes in the first Altair), and front panel switches, and you'd have a working computer. Being able to have your own computer was so exciting that there were plenty of people who wanted to buy them, however limited.

Web pages weren't designed to be a UI for applications, but they're just good enough. And for a significant number of users, software that you can use from any browser will be enough of a win in itself to outweigh any awkwardness in the UI. Maybe you can't write the best-looking spreadsheet using HTML, but you can write a spreadsheet that several people can use simultaneously from different locations without special client software, or that can incorporate live data feeds, or that can page you when certain conditions are triggered. More importantly, you can write new kinds of applications that don't even have names yet. VisiCalc was not merely a microcomputer version of a mainframe application, after all-- it was a new type of application.

Of course, server-based applications don't have to be Web-based. You could have some other kind of client. But I'm pretty sure that's a bad idea. It would be very convenient if you could assume that everyone would install your client-- so convenient that you could easily convince yourself that they all would-- but if they don't, you're hosed. Because Web-based software assumes nothing about the client, it will work anywhere the Web works. That's a big advantage already, and the advantage will grow as new Web devices proliferate. Users will like you because your software just works, and your life will be easier because you won't have to tweak it for every new client. [16]

I feel like I've watched the evolution of the Web as closely as anyone, and I can't

predict what's going to happen with clients. Convergence is probably coming, but where? I can't pick a winner. One thing I can predict is conflict between AOL and Microsoft. Whatever Microsoft's .NET turns out to be, it will probably involve connecting the desktop to servers. Unless AOL fights back, they will either be pushed aside or turned into a pipe between Microsoft client and server software. If Microsoft and AOL get into a client war, the only thing sure to work on both will be browsing the Web, meaning Web-based applications will be the only kind that work everywhere.

How will it all play out? I don't know. And you don't have to know if you bet on Web-based applications. No one can break that without breaking browsing. The Web may not be the only way to deliver software, but it's one that works now and will continue to work for a long time. Web-based applications are cheap to develop, and easy for even the smallest startup to deliver. They're a lot of work, and of a particularly stressful kind, but that only makes the odds better for startups.

Why Not?

E. B. White was amused to learn from a farmer friend that many electrified fences don't have any current running through them. The cows apparently learn to stay away from them, and after that you don't need the current. "Rise up, cows!" he wrote, "Take your liberty while despots snore!"

If you're a hacker who has thought of one day starting a startup, there are probably two things keeping you from doing it. One is that you don't know anything about business. The other is that you're afraid of competition. Neither of these fences have any current in them.

There are only two things you have to know about business: build something users love, and make more than you spend. If you get these two right, you'll be ahead of most startups. You can figure out the rest as you go.

You may not at first make more than you spend, but as long as the gap is closing fast enough you'll be ok. If you start out underfunded, it will at least encourage a habit of frugality. The less you spend, the easier it is to make more than you spend. Fortunately, it can be very cheap to launch a Web-based application. We launched on under $10,000, and it would be even cheaper today. We had to spend thousands on a server, and thousands more to get SSL. (The only company selling SSL software at the time was Netscape.) Now you can rent a much more powerful server, with SSL included, for less than we paid for bandwidth alone. You could launch a Web-based application now for less than the cost of a fancy office chair.

As for building something users love, here are some general tips. Start by making something clean and simple that you would want to use yourself. Get a version 1.0 out fast, then continue to improve the software, listening closely to the users as you do. The customer is always right, but different customers are right about different things; the least sophisticated users show you what you need to simplify and clarify, and the most sophisticated tell you what features you need to add. The

best thing software can be is easy, but the way to do this is to get the defaults right, not to limit users' choices. Don't get complacent if your competitors' software is lame; the standard to compare your software to is what it could be, not what your current competitors happen to have. Use your software yourself, all the time. Viaweb was supposed to be an online store builder, but we used it to make our own site too. Don't listen to marketing people or designers or product managers just because of their job titles. If they have good ideas, use them, but it's up to you to decide; software has to be designed by hackers who understand design, not designers who know a little about software. If you can't design software as well as implement it, don't start a startup.

Now let's talk about competition. What you're afraid of is not presumably groups of hackers like you, but actual companies, with offices and business plans and salesmen and so on, right? Well, they are more afraid of you than you are of them, and they're right. It's a lot easier for a couple of hackers to figure out how to rent office space or hire sales people than it is for a company of any size to get software written. I've been on both sides, and I know. When Viaweb was bought by Yahoo, I suddenly found myself working for a big company, and it was like trying to run through waist-deep water.

I don't mean to disparage Yahoo. They had some good hackers, and the top management were real butt-kickers. For a big company, they were exceptional. But they were still only about a tenth as productive as a small startup. No big company can do much better than that. What's scary about Microsoft is that a company so big can develop software at all. They're like a mountain that can walk.

Don't be intimidated. You can do as much that Microsoft can't as they can do that you can't. And no one can stop you. You don't have to ask anyone's permission to develop Web-based applications. You don't have to do licensing deals, or get shelf space in retail stores, or grovel to have your application bundled with the OS. You can deliver software right to the browser, and no one can get between you and potential users without preventing them from browsing the Web.

You may not believe it, but I promise you, Microsoft is scared of you. The complacent middle managers may not be, but Bill is, because he was you once, back in 1975, the last time a new way of delivering software appeared.

Notes

[1] Realizing that much of the money is in the services, companies building lightweight clients have usually tried to combine the hardware with an online service. This approach has not worked well, partly because you need two different kinds of companies to build consumer electronics and to run an online service, and partly because users hate the idea. Giving away the razor and making money on

the blades may work for Gillette, but a razor is much smaller commitment than a Web terminal. Cell phone handset makers are satisfied to sell hardware without trying to capture the service revenue as well. That should probably be the model for Internet clients too. If someone just sold a nice-looking little box with a Web browser that you could use to connect through any ISP, every technophobe in the country would buy one.

[2] Security always depends more on not screwing up than any design decision, but the nature of server-based software will make developers pay more attention to not screwing up. Compromising a server could cause such damage that ASPs (that want to stay in business) are likely to be careful about security.

[3] In 1995, when we started Viaweb, Java applets were supposed to be the technology everyone was going to use to develop server-based applications. Applets seemed to us an old-fashioned idea. Download programs to run on the client? Simpler just to go all the way and run the programs on the server. We wasted little time on applets, but countless other startups must have been lured into this tar pit. Few can have escaped alive, or Microsoft could not have gotten away with dropping Java in the most recent version of Explorer.

[4] This point is due to Trevor Blackwell, who adds "the cost of writing software goes up more than linearly with its size. Perhaps this is mainly due to fixing old bugs, and the cost can be more linear if all bugs are found quickly."

[5] The hardest kind of bug to find may be a variant of compound bug where one bug happens to compensate for another. When you fix one bug, the other becomes visible. But it will seem as if the fix is at fault, since that was the last thing you changed.

[6] Within Viaweb we once had a contest to describe the worst thing about our software. Two customer support people tied for first prize with entries I still shiver to recall. We fixed both problems immediately.

[7] Robert Morris wrote the ordering system, which shoppers used to place orders. Trevor Blackwell wrote the image generator and the manager, which merchants used to retrieve orders, view statistics, and configure domain names etc. I wrote the editor, which merchants used to build their sites. The ordering system and image generator were written in C and C++, the manager mostly in Perl, and the editor in Lisp.

[8] Price discrimination is so pervasive (how often have you heard a retailer claim that their buying power meant lower prices for you?) that I was surprised to find it was outlawed in the U.S. by the Robinson-Patman Act of 1936. This law does not appear to be vigorously enforced.

[9] In *No Logo,* Naomi Klein says that clothing brands favored by "urban youth" do not try too hard to prevent shoplifting because in their target market the shoplifters are also the fashion leaders.

[10] Companies often wonder what to outsource and what not to. One possible answer: outsource any job that's not directly exposed to competitive pressure, because outsourcing it will thereby expose it to competitive pressure.

[11] The two guys were Dan Bricklin and Bob Frankston. Dan wrote a prototype in Basic in a couple days, then over the course of the next year they worked together (mostly at night) to make a more powerful version written in 6502 machine language. Dan was at Harvard Business School at the time and Bob nominally had a day job writing software. "There was no great risk in doing a business," Bob wrote, "If it failed it failed. No big deal."

[12] It's not quite as easy as I make it sound. It took a painfully long time for word of mouth to get going, and we did not start to get a lot of press coverage until we hired a PR firm (admittedly the best in the business) for $16,000 per month. However, it was true that the only significant channel was our own Web site.

[13] If the Mac was so great, why did it lose? Cost, again. Microsoft concentrated on the software business, and unleashed a swarm of cheap component suppliers on Apple hardware. It did not help, either, that suits took over during a critical period.

[14] One thing that would help Web-based applications, and help keep the next generation of software from being overshadowed by Microsoft, would be a good open-source browser. Mozilla is open-source but seems to have suffered from having been corporate software for so long. A small, fast browser that was actively maintained would be a great thing in itself, and would probably also encourage companies to build little Web appliances.

Among other things, a proper open-source browser would cause HTTP and HTML to continue to evolve (as e.g. Perl has). It would help Web-based applications greatly to be able to distinguish between selecting a link and following it; all you'd need to do this would be a trivial enhancement of HTTP, to allow multiple urls in a request. Cascading menus would also be good.

If you want to change the world, write a new Mosaic. Think it's too late? In 1998 a lot of people thought it was too late to launch a new search engine, but Google proved them wrong. There is always room for something new if the current options suck enough. Make sure it works on all the free OSes first-- new things start with their users.

[15] Trevor Blackwell, who probably knows more about this from personal experience than anyone, writes:

"I would go farther in saying that because server-based software is so hard on the programmers, it causes a fundamental economic shift away from large companies. It requires the kind of intensity and dedication from programmers that they will only be willing to provide when it's their own company. Software companies can

hire skilled people to work in a not-too-demanding environment, and can hire unskilled people to endure hardships, but they can't hire highly skilled people to bust their asses. Since capital is no longer needed, big companies have little to bring to the table."

[16] In the original version of this essay, I advised avoiding Javascript. That was a good plan in 2001, but Javascript now works.

Thanks to Sarah Harlin, Trevor Blackwell, Robert Morris, Eric Raymond, Ken Anderson, and Dan Giffin for reading drafts of this paper; to Dan Bricklin and Bob Frankston for information about VisiCalc; and again to Ken Anderson for inviting me to speak at BBN.

You'll find this essay and 14 others in ***Hackers & Painters***.

▪

Some Technical Details

▪

Japanese Translation

▪

Microsoft finally agrees

▪

Gates Email

What Made Lisp Different

December 2001 (rev. May 2002)

(This article came about in response to some questions on the [LL1](#) mailing list. It is now incorporated in [Revenge of the Nerds](#).)

When McCarthy designed Lisp in the late 1950s, it was a radical departure from existing languages, the most important of which was [Fortran](#).

Lisp embodied nine new ideas:

1. Conditionals. A conditional is an if-then-else construct. We take these for granted now. They were [invented](#) by McCarthy in the course of developing Lisp. (Fortran at that time only had a conditional goto, closely based on the branch instruction in the underlying hardware.) McCarthy, who was on the Algol committee, got conditionals into Algol, whence they spread to most other languages.

2. A function type. In Lisp, functions are first class objects-- they're a data type just like integers, strings, etc, and have a literal representation, can be stored in variables, can be passed as arguments, and so on.

3. Recursion. Recursion existed as a mathematical concept before Lisp of course, but Lisp was the first programming language to support it. (It's arguably implicit in making functions first class objects.)

4. A new concept of variables. In Lisp, all variables are effectively pointers. Values are what have types, not variables, and assigning or binding variables means copying pointers, not what they point to.

5. Garbage-collection.

6. Programs composed of expressions. Lisp programs are trees of expressions, each of which returns a value. (In some Lisps expressions can return multiple values.) This is in contrast to Fortran and most succeeding languages, which distinguish between expressions and statements.

It was natural to have this distinction in Fortran because (not surprisingly in a language where the input format was punched cards) the language was line-oriented. You could not nest statements. And so while you needed expressions for

math to work, there was no point in making anything else return a value, because there could not be anything waiting for it.

This limitation went away with the arrival of block-structured languages, but by then it was too late. The distinction between expressions and statements was entrenched. It spread from Fortran into Algol and thence to both their descendants.

When a language is made entirely of expressions, you can compose expressions however you want. You can say either (using Arc syntax)

(if foo (= x 1) (= x 2))

or

(= x (if foo 1 2))

7. A symbol type. Symbols differ from strings in that you can test equality by comparing a pointer.

8. A notation for code using trees of symbols.

9. The whole language always available. There is no real distinction between read-time, compile-time, and runtime. You can compile or run code while reading, read or run code while compiling, and read or compile code at runtime.

Running code at read-time lets users reprogram Lisp's syntax; running code at compile-time is the basis of macros; compiling at runtime is the basis of Lisp's use as an extension language in programs like Emacs; and reading at runtime enables programs to communicate using s-expressions, an idea recently reinvented as XML.

When Lisp was first invented, all these ideas were far removed from ordinary programming practice, which was dictated largely by the hardware available in the late 1950s.

Over time, the default language, embodied in a succession of popular languages, has gradually evolved toward Lisp. 1-5 are now widespread. 6 is starting to appear in the mainstream. Python has a form of 7, though there doesn't seem to be any syntax for it. 8, which (with 9) is what makes Lisp macros possible, is so far still unique to Lisp, perhaps because (a) it requires those parens, or something just as bad, and (b) if you add that final increment of power, you can no longer claim to have invented a new language, but only to have designed a new dialect of Lisp ; -)

Though useful to present-day programmers, it's strange to describe Lisp in terms of its variation from the random expedients other languages adopted. That was not, probably, how McCarthy thought of it. Lisp wasn't designed to fix the mistakes in Fortran; it came about more as the byproduct of an attempt to axiomatize

[computation](#).

.

[Japanese Translation](#)

Why Arc Isn't Especially Object-Oriented

There is a kind of mania for object-oriented programming at the moment, but some of the [smartest programmers](#) I know are some of the least excited about it.

My own feeling is that object-oriented programming is a useful technique in some cases, but it isn't something that has to pervade every program you write. You should be able to define new types, but you shouldn't have to express every program as the definition of new types.

I think there are five reasons people like object-oriented programming, and three and a half of them are bad:

1. Object-oriented programming is exciting if you have a statically-typed language without lexical closures or macros. To some degree, it offers a way around these limitations. (See [Greenspun's Tenth Rule](#).)

2. Object-oriented programming is popular in big companies, because it suits the way they write software. At big companies, software tends to be written by large (and frequently changing) teams of mediocre programmers. Object-oriented programming imposes a discipline on these programmers that prevents any one of them from doing too much damage. The price is that the resulting code is bloated with protocols and full of duplication. This is not too high a price for big companies, because their software is probably going to be bloated and full of duplication anyway.

3. Object-oriented programming generates a lot of what looks like work. Back in the days of fanfold, there was a type of programmer who would only put five or ten lines of code on a page, preceded by twenty lines of elaborately formatted comments. Object-oriented programming is like crack for these people: it lets you incorporate all this scaffolding right into your source code. Something that a Lisp hacker might handle by pushing a symbol onto a list becomes a whole file of classes and methods. So it is a good tool if you want to convince yourself, or someone else, that you are doing a lot of work.

4. If a language is itself an object-oriented program, it can be extended by users. Well, maybe. Or maybe you can do even better by offering the sub-concepts of object-oriented programming a la carte. Overloading, for example, is not intrinsically tied to classes. We'll see.

5. Object-oriented abstractions map neatly onto the domains of certain specific kinds of programs, like simulations and CAD systems.

I personally have never needed object-oriented abstractions. Common Lisp has an enormously powerful object system and I've never used it once. I've done a lot of things (e.g. making hash tables full of closures) that would have required object-oriented techniques to do in wimpier languages, but I have never had to use CLOS.

Maybe I'm just stupid, or have worked on some limited subset of applications. There is a danger in designing a language based on one's own experience of programming. But it seems more dangerous to put stuff in that you've never needed because it's thought to be a good idea.

▪ [Rees Re: OO](#)

▪ [Spanish Translation](#)

Taste for Makers

February 2002

"...Copernicus' aesthetic objections to [equants] provided one essential motive for his rejection of the Ptolemaic system...."

- Thomas Kuhn, *The Copernican Revolution*

"All of us had been trained by Kelly Johnson and believed fanatically in his insistence that an airplane that looked beautiful would fly the same way."

- Ben Rich, *Skunk Works*

"Beauty is the first test: there is no permanent place in this world for ugly mathematics."

- G. H. Hardy, *A Mathematician's Apology*

I was talking recently to a friend who teaches at MIT. His field is hot now and every year he is inundated by applications from would-be graduate students. "A lot of them seem smart," he said. "What I can't tell is whether they have any kind of taste."

Taste. You don't hear that word much now. And yet we still need the underlying concept, whatever we call it. What my friend meant was that he wanted students who were not just good technicians, but who could use their technical knowledge to design beautiful things.

Mathematicians call good work "beautiful," and so, either now or in the past, have scientists, engineers, musicians, architects, designers, writers, and painters. Is it just a coincidence that they used the same word, or is there some overlap in what they meant? If there is an overlap, can we use one field's discoveries about beauty to help us in another?

For those of us who design things, these are not just theoretical questions. If there is such a thing as beauty, we need to be able to recognize it. We need good taste to make good things. Instead of treating beauty as an airy abstraction, to be either blathered about or avoided depending on how one feels about airy abstractions, let's try considering it as a practical question: *how do you make good stuff?*

If you mention taste nowadays, a lot of people will tell you that "taste is subjective." They believe this because it really feels that way to them. When they like something, they have no idea why. It could be because it's beautiful, or because their mother had one, or because they saw a movie star with one in a magazine, or because they know it's expensive. Their thoughts are a tangle of unexamined impulses.

Most of us are encouraged, as children, to leave this tangle unexamined. If you make fun of your little brother for coloring people green in his coloring book, your mother is likely to tell you something like "you like to do it your way and he likes to do it his way."

Your mother at this point is not trying to teach you important truths about aesthetics. She's trying to get the two of you to stop bickering.

Like many of the half-truths adults tell us, this one contradicts other things they tell us. After dinning into you that taste is merely a matter of personal preference, they take you to the museum and tell you that you should pay attention because Leonardo is a great artist.

What goes through the kid's head at this point? What does he think "great artist" means? After having been told for years that everyone just likes to do things their own way, he is unlikely to head straight for the conclusion that a great artist is someone whose work is *better* than the others'. A far more likely theory, in his Ptolemaic model of the universe, is that a great artist is something that's good for you, like broccoli, because someone said so in a book.

Saying that taste is just personal preference is a good way to prevent disputes. The trouble is, it's not true. You feel this when you start to design things.

Whatever job people do, they naturally want to do better. Football players like to win games. CEOs like to increase earnings. It's a matter of pride, and a real pleasure, to get better at your job. But if your job is to design things, and there is no such thing as beauty, then there is *no way to get better at your job.* If taste is just personal preference, then everyone's is already perfect: you like whatever you like, and that's it.

As in any job, as you continue to design things, you'll get better at it. Your tastes will change. And, like anyone who gets better at their job, you'll know you're getting better. If so, your old tastes were not merely different, but worse. Poof goes the axiom that taste can't be wrong.

Relativism is fashionable at the moment, and that may hamper you from thinking about taste, even as yours grows. But if you come out of the closet and admit, at least to yourself, that there is such a thing as good and bad design, then you can start to study good design in detail. How has your taste changed? When you made mistakes, what caused you to make them? What have other people learned about design?

Once you start to examine the question, it's surprising how much different fields' ideas of beauty have in common. The same principles of good design crop up again and again.

Good design is simple. You hear this from math to painting. In math it means that a shorter proof tends to be a better one. Where axioms are concerned, especially, less is more. It means much the same thing in programming. For architects and designers it means that beauty should depend on a few carefully chosen structural elements rather than a profusion of superficial ornament. (Ornament is not in itself bad, only when it's camouflage on insipid form.) Similarly, in painting, a still life of a few carefully observed and solidly modelled objects will tend to be more interesting than a stretch of flashy but mindlessly repetitive painting of, say, a lace collar. In writing it means: say what you mean and say it briefly.

It seems strange to have to emphasize simplicity. You'd think simple would be the default. Ornate is more work. But something seems to come over people when they try to be creative. Beginning writers adopt a pompous tone that doesn't sound anything like the way they speak. Designers trying to be artistic resort to swooshes and curlicues. Painters discover that they're expressionists. It's all evasion. Underneath the long words or the "expressive" brush strokes, there is not much going on, and that's frightening.

When you're forced to be simple, you're forced to face the real problem. When you can't deliver ornament, you have to deliver substance.

Good design is timeless. In math, every proof is timeless unless it contains a mistake. So what does Hardy mean when he says there is no permanent place for ugly mathematics? He means the same thing Kelly Johnson did: if something is ugly, it can't be the best solution. There must be a better one, and eventually someone will discover it.

Aiming at timelessness is a way to make yourself find the best answer: if you can imagine someone surpassing you, you should do it yourself. Some of the greatest masters did this so well that they left little room for those who came after. Every engraver since Durer has had to live in his shadow.

Aiming at timelessness is also a way to evade the grip of fashion. Fashions almost by definition change with time, so if you can make something that will still look good far into the future, then its appeal must derive more from merit and less from fashion.

Strangely enough, if you want to make something that will appeal to future generations, one way to do it is to try to appeal to past generations. It's hard to guess what the future will be like, but we can be sure it will be like the past in caring nothing for present fashions. So if you can make something that appeals to people today and would also have appealed to people in 1500, there is a good chance it will appeal to people in 2500.

Good design solves the right problem. The typical stove has four burners arranged in a square, and a dial to control each. How do you arrange the dials? The simplest answer is to put them in a row. But this is a simple answer to the wrong question. The dials are for humans to use, and if you put them in a row, the unlucky human will have to stop and think each time about which dial matches which burner. Better to arrange the dials in a square like the burners.

A lot of bad design is industrious, but misguided. In the mid twentieth century there was a vogue for setting text in sans-serif fonts. These fonts *are* closer to the pure, underlying letterforms. But in text that's not the problem you're trying to solve. For legibility it's more important that letters be easy to tell apart. It may look Victorian, but a Times Roman lowercase g is easy to tell from a lowercase y.

Problems can be improved as well as solutions. In software, an intractable problem can usually be replaced by an equivalent one that's easy to solve. Physics progressed faster as the problem became predicting observable behavior, instead of reconciling it with scripture.

Good design is suggestive. Jane Austen's novels contain almost no description; instead of telling you how everything looks, she tells her story so well that you envision the scene for yourself. Likewise, a painting that suggests is usually more engaging than one that tells. Everyone makes up their own story about the Mona Lisa.

In architecture and design, this principle means that a building or object should let you use it how you want: a good building, for example, will serve as a backdrop for whatever life people want to lead in it, instead of making them live as if they were

executing a program written by the architect.

In software, it means you should give users a few basic elements that they can combine as they wish, like Lego. In math it means a proof that becomes the basis for a lot of new work is preferable to a proof that was difficult, but doesn't lead to future discoveries; in the sciences generally, citation is considered a rough indicator of merit.

Good design is often slightly funny. This one may not always be true. But Durer's engravings and Saarinen's womb chair and the Pantheon and the original Porsche 911 all seem to me slightly funny. Godel's incompleteness theorem seems like a practical joke.

I think it's because humor is related to strength. To have a sense of humor is to be strong: to keep one's sense of humor is to shrug off misfortunes, and to lose one's sense of humor is to be wounded by them. And so the mark-- or at least the prerogative-- of strength is not to take oneself too seriously. The confident will often, like swallows, seem to be making fun of the whole process slightly, as Hitchcock does in his films or Bruegel in his paintings-- or Shakespeare, for that matter.

Good design may not have to be funny, but it's hard to imagine something that could be called humorless also being good design.

Good design is hard. If you look at the people who've done great work, one thing they all seem to have in common is that they worked very hard. If you're not working hard, you're probably wasting your time.

Hard problems call for great efforts. In math, difficult proofs require ingenious solutions, and those tend to be interesting. Ditto in engineering.

When you have to climb a mountain you toss everything unnecessary out of your pack. And so an architect who has to build on a difficult site, or a small budget, will find that he is forced to produce an elegant design. Fashions and flourishes get knocked aside by the difficult business of solving the problem at all.

Not every kind of hard is good. There is good pain and bad pain. You want the kind of pain you get from going running, not the kind you get from stepping on a nail. A difficult problem could be good for a designer, but a fickle client or unreliable materials would not be.

In art, the highest place has traditionally been given to paintings of people. There is something to this tradition, and not just because pictures of faces get to press buttons in our brains that other pictures don't. We are so good at looking at faces

that we force anyone who draws them to work hard to satisfy us. If you draw a tree and you change the angle of a branch five degrees, no one will know. When you change the angle of someone's eye five degrees, people notice.

When Bauhaus designers adopted Sullivan's "form follows function," what they meant was, form *should* follow function. And if function is hard enough, form is forced to follow it, because there is no effort to spare for error. Wild animals are beautiful because they have hard lives.

Good design looks easy. Like great athletes, great designers make it look easy. Mostly this is an illusion. The easy, conversational tone of good writing comes only on the eighth rewrite.

In science and engineering, some of the greatest discoveries seem so simple that you say to yourself, I could have thought of that. The discoverer is entitled to reply, why didn't you?

Some Leonardo heads are just a few lines. You look at them and you think, all you have to do is get eight or ten lines in the right place and you've made this beautiful portrait. Well, yes, but you have to get them in *exactly* the right place. The slightest error will make the whole thing collapse.

Line drawings are in fact the most difficult visual medium, because they demand near perfection. In math terms, they are a closed-form solution; lesser artists literally solve the same problems by successive approximation. One of the reasons kids give up drawing at ten or so is that they decide to start drawing like grownups, and one of the first things they try is a line drawing of a face. Smack!

In most fields the appearance of ease seems to come with practice. Perhaps what practice does is train your unconscious mind to handle tasks that used to require conscious thought. In some cases you literally train your body. An expert pianist can play notes faster than the brain can send signals to his hand. Likewise an artist, after a while, can make visual perception flow in through his eye and out through his hand as automatically as someone tapping his foot to a beat.

When people talk about being in "the zone," I think what they mean is that the spinal cord has the situation under control. Your spinal cord is less hesitant, and it frees conscious thought for the hard problems.

Good design uses symmetry. I think symmetry may just be one way to achieve simplicity, but it's important enough to be mentioned on its own. Nature uses it a lot, which is a good sign.

There are two kinds of symmetry, repetition and recursion. Recursion means

repetition in subelements, like the pattern of veins in a leaf.

Symmetry is unfashionable in some fields now, in reaction to excesses in the past. Architects started consciously making buildings asymmetric in Victorian times and by the 1920s asymmetry was an explicit premise of modernist architecture. Even these buildings only tended to be asymmetric about major axes, though; there were hundreds of minor symmetries.

In writing you find symmetry at every level, from the phrases in a sentence to the plot of a novel. You find the same in music and art. Mosaics (and some Cezannes) get extra visual punch by making the whole picture out of the same atoms. Compositional symmetry yields some of the most memorable paintings, especially when two halves react to one another, as in the *Creation of Adam* or *American Gothic.*

In math and engineering, recursion, especially, is a big win. Inductive proofs are wonderfully short. In software, a problem that can be solved by recursion is nearly always best solved that way. The Eiffel Tower looks striking partly because it is a recursive solution, a tower on a tower.

The danger of symmetry, and repetition especially, is that it can be used as a substitute for thought.

Good design resembles nature. It's not so much that resembling nature is intrinsically good as that nature has had a long time to work on the problem. It's a good sign when your answer resembles nature's.

It's not cheating to copy. Few would deny that a story should be like life. Working from life is a valuable tool in painting too, though its role has often been misunderstood. The aim is not simply to make a record. The point of painting from life is that it gives your mind something to chew on: when your eyes are looking at something, your hand will do more interesting work.

Imitating nature also works in engineering. Boats have long had spines and ribs like an animal's ribcage. In some cases we may have to wait for better technology: early aircraft designers were mistaken to design aircraft that looked like birds, because they didn't have materials or power sources light enough (the Wrights' engine weighed 152 lbs. and generated only 12 hp.) or control systems sophisticated enough for machines that flew like birds, but I could imagine little unmanned reconnaissance planes flying like birds in fifty years.

Now that we have enough computer power, we can imitate nature's method as well as its results. Genetic algorithms may let us create things too complex to design in the ordinary sense.

Good design is redesign. It's rare to get things right the first time. Experts expect to throw away some early work. They plan for plans to change.

It takes confidence to throw work away. You have to be able to think, *there's more where that came from.* When people first start drawing, for example, they're often reluctant to redo parts that aren't right; they feel they've been lucky to get that far, and if they try to redo something, it will turn out worse. Instead they convince themselves that the drawing is not that bad, really-- in fact, maybe they meant it to look that way.

Dangerous territory, that; if anything you should cultivate dissatisfaction. In Leonardo's [drawings](#) there are often five or six attempts to get a line right. The distinctive back of the Porsche 911 only appeared in the redesign of an awkward [prototype](#). In Wright's early plans for the [Guggenheim](#), the right half was a ziggurat; he inverted it to get the present shape.

Mistakes are natural. Instead of treating them as disasters, make them easy to acknowledge and easy to fix. Leonardo more or less invented the sketch, as a way to make drawing bear a greater weight of exploration. Open-source software has fewer bugs because it admits the possibility of bugs.

It helps to have a medium that makes change easy. When oil paint replaced tempera in the fifteenth century, it helped painters to deal with difficult subjects like the human figure because, unlike tempera, oil can be blended and overpainted.

Good design can copy. Attitudes to copying often make a round trip. A novice imitates without knowing it; next he tries consciously to be original; finally, he decides it's more important to be right than original.

Unknowing imitation is almost a recipe for bad design. If you don't know where your ideas are coming from, you're probably imitating an imitator. Raphael so pervaded mid-nineteenth century taste that almost anyone who tried to draw was imitating him, often at several removes. It was this, more than Raphael's own work, that bothered the Pre-Raphaelites.

The ambitious are not content to imitate. The second phase in the growth of taste is a conscious attempt at originality.

I think the greatest masters go on to achieve a kind of selflessness. They just want to get the right answer, and if part of the right answer has already been discovered by someone else, that's no reason not to use it. They're confident enough to take from anyone without feeling that their own vision will be lost in the process.

Good design is often strange. Some of the very best work has an uncanny quality: [Euler's Formula](#), Bruegel's *Hunters in the Snow,* the [SR-71](#), [Lisp](#). They're not just beautiful, but strangely beautiful.

I'm not sure why. It may just be my own stupidity. A can-opener must seem miraculous to a dog. Maybe if I were smart enough it would seem the most natural thing in the world that $e^{i*pi} = -1$. It is after all necessarily true.

Most of the qualities I've mentioned are things that can be cultivated, but I don't think it works to cultivate strangeness. The best you can do is not squash it if it starts to appear. Einstein didn't try to make relativity strange. He tried to make it true, and the truth turned out to be strange.

At an art school where I once studied, the students wanted most of all to develop a personal style. But if you just try to make good things, you'll inevitably do it in a distinctive way, just as each person walks in a distinctive way. Michelangelo was not trying to paint like Michelangelo. He was just trying to paint well; he couldn't help painting like Michelangelo.

The only style worth having is the one you can't help. And this is especially true for strangeness. There is no shortcut to it. The Northwest Passage that the Mannerists, the Romantics, and two generations of American high school students have searched for does not seem to exist. The only way to get there is to go through good and come out the other side.

Good design happens in chunks. The inhabitants of fifteenth century Florence included Brunelleschi, Ghiberti, Donatello, Masaccio, Filippo Lippi, Fra Angelico, Verrocchio, Botticelli, Leonardo, and Michelangelo. Milan at the time was as big as Florence. How many fifteenth century Milanese artists can you name?

Something was happening in Florence in the fifteenth century. And it can't have been heredity, because it isn't happening now. You have to assume that whatever inborn ability Leonardo and Michelangelo had, there were people born in Milan with just as much. What happened to the Milanese Leonardo?

There are roughly a thousand times as many people alive in the US right now as lived in Florence during the fifteenth century. A thousand Leonardos and a thousand Michelangelos walk among us. If DNA ruled, we should be greeted daily by artistic marvels. We aren't, and the reason is that to make Leonardo you need more than his innate ability. You also need Florence in 1450.

Nothing is more powerful than a community of talented people working on related problems. Genes count for little by comparison: being a genetic Leonardo was not enough to compensate for having been born near Milan instead of Florence. Today

we move around more, but great work still comes disproportionately from a few hotspots: the Bauhaus, the Manhattan Project, the *New Yorker,* Lockheed's Skunk Works, Xerox Parc.

At any given time there are a few hot topics and a few groups doing great work on them, and it's nearly impossible to do good work yourself if you're too far removed from one of these centers. You can push or pull these trends to some extent, but you can't break away from them. (Maybe *you* can, but the Milanese Leonardo couldn't.)

Good design is often daring. At every period of history, people have believed things that were just ridiculous, and believed them so strongly that you risked ostracism or even violence by saying otherwise.

If our own time were any different, that would be remarkable. As far as I can tell it isn't.

This problem afflicts not just every era, but in some degree every field. Much Renaissance art was in its time considered shockingly secular: according to Vasari, Botticelli repented and gave up painting, and Fra Bartolommeo and Lorenzo di Credi actually burned some of their work. Einstein's theory of relativity offended many contemporary physicists, and was not fully accepted for decades-- in France, not until the 1950s.

Today's experimental error is tomorrow's new theory. If you want to discover great new things, then instead of turning a blind eye to the places where conventional wisdom and truth don't quite meet, you should pay particular attention to them.

As a practical matter, I think it's easier to see ugliness than to imagine beauty. Most of the people who've made beautiful things seem to have done it by fixing something that they thought ugly. Great work usually seems to happen because someone sees something and thinks, *I could do better than that.* Giotto saw traditional Byzantine madonnas painted according to a formula that had satisfied everyone for centuries, and to him they looked wooden and unnatural. Copernicus was so troubled by a hack that all his contemporaries could tolerate that he felt there must be a better solution.

Intolerance for ugliness is not in itself enough. You have to understand a field well before you develop a good nose for what needs fixing. You have to do your homework. But as you become expert in a field, you'll start to hear little voices saying, *What a hack! There must be a better way.* Don't ignore those voices. Cultivate them. The recipe for great work is: very exacting taste, plus the ability to gratify it.

Notes

Sullivan actually said "form ever follows function," but I think the usual misquotation is closer to what modernist architects meant.

Stephen G. Brush, "Why was Relativity Accepted?" *Phys. Perspect. 1 (1999) 184-214.*

.

Japanese Translation

.

Chinese Translation

.

Slovenian Translation

.

German Translation

.

Interview: Milton Glaser

.

Russian Translation

You'll find this essay and 14 others in **_Hackers & Painters_**.

What Languages Fix

Kevin Kelleher suggested an interesting way to compare programming languages: to describe each in terms of the problem it fixes. The surprising thing is how many, and how well, languages can be described this way.

Algol: Assembly language is too low-level.

Pascal: Algol doesn't have enough data types.

Modula: Pascal is too wimpy for systems programming.

Simula: Algol isn't good enough at simulations.

Smalltalk: Not everything in Simula is an object.

Fortran: Assembly language is too low-level.

Cobol: Fortran is scary.

PL/1: Fortran doesn't have enough data types.

Ada: Every existing language is missing something.

Basic: Fortran is scary.

APL: Fortran isn't good enough at manipulating arrays.

J: APL requires its own character set.

C: Assembly language is too low-level.

C++: C is too low-level.

Java: C++ is a kludge. And Microsoft is going to crush us.

C#: Java is controlled by Sun.

Lisp: Turing Machines are an awkward way to describe computation.

Scheme: MacLisp is a kludge.

T: Scheme has no libraries.

Common Lisp: There are too many dialects of Lisp.

Dylan: Scheme has no libraries, and Lisp syntax is scary.

Perl: Shell scripts/awk/sed are not enough like programming languages.

Python: Perl is a kludge.

Ruby: Perl is a kludge, and Lisp syntax is scary.

Prolog: Programming is not enough like logic.

.

[Japanese Translation](#)

.

[French Translation](#)

.

[Portuguese Translation](#)

Succinctness is Power

May 2002

"The quantity of meaning compressed into a small space by algebraic signs, is another circumstance that facilitates the reasonings we are accustomed to carry on by their aid."

- Charles Babbage, quoted in Iverson's Turing Award Lecture

In the discussion about issues raised by [Revenge of the Nerds](#) on the LL1 mailing list, Paul Prescod wrote something that stuck in my mind.

> Python's goal is regularity and readability, not succinctness.

On the face of it, this seems a rather damning thing to claim about a programming language. As far as I can tell, succinctness = power. If so, then substituting, we get

> Python's goal is regularity and readability, not power.

and this doesn't seem a tradeoff (if it *is* a tradeoff) that you'd want to make. It's not far from saying that Python's goal is not to be effective as a programming language.

Does succinctness = power? This seems to me an important question, maybe the most important question for anyone interested in language design, and one that it would be useful to confront directly. I don't feel sure yet that the answer is a simple yes, but it seems a good hypothesis to begin with.

Hypothesis

My hypothesis is that succinctness is power, or is close enough that except in pathological examples you can treat them as identical.

It seems to me that succinctness is what programming languages are *for.* Computers would be just as happy to be told what to do directly in machine language. I think that the main reason we take the trouble to develop high-level languages is to get leverage, so that we can say (and more importantly, think) in 10 lines of a high-level language what would require 1000 lines of machine language. In other words, the main point of high-level languages is to make source

code smaller.

If smaller source code is the purpose of high-level languages, and the power of something is how well it achieves its purpose, then the measure of the power of a programming language is how small it makes your programs.

Conversely, a language that doesn't make your programs small is doing a bad job of what programming languages are supposed to do, like a knife that doesn't cut well, or printing that's illegible.

Metrics

Small in what sense though? The most common measure of code size is lines of code. But I think that this metric is the most common because it is the easiest to measure. I don't think anyone really believes it is the true test of the length of a program. Different languages have different conventions for how much you should put on a line; in C a lot of lines have nothing on them but a delimiter or two.

Another easy test is the number of characters in a program, but this is not very good either; some languages (Perl, for example) just use shorter identifiers than others.

I think a better measure of the size of a program would be the number of elements, where an element is anything that would be a distinct node if you drew a tree representing the source code. The name of a variable or function is an element; an integer or a floating-point number is an element; a segment of literal text is an element; an element of a pattern, or a format directive, is an element; a new block is an element. There are borderline cases (is -5 two elements or one?) but I think most of them are the same for every language, so they don't affect comparisons much.

This metric needs fleshing out, and it could require interpretation in the case of specific languages, but I think it tries to measure the right thing, which is the number of parts a program has. I think the tree you'd draw in this exercise is what you have to make in your head in order to conceive of the program, and so its size is proportionate to the amount of work you have to do to write or read it.

Design

This kind of metric would allow us to compare different languages, but that is not, at least for me, its main value. The main value of the succinctness test is as a guide in *designing* languages. The most useful comparison between languages is between two potential variants of the same language. What can I do in the language to make programs shorter?

If the conceptual load of a program is proportionate to its complexity, and a given programmer can tolerate a fixed conceptual load, then this is the same as asking, what can I do to enable programmers to get the most done? And that seems to me

identical to asking, how can I design a good language?

(Incidentally, nothing makes it more patently obvious that the old chestnut "all languages are equivalent" is false than designing languages. When you are designing a new language, you're *constantly* comparing two languages-- the language if I did x, and if I didn't-- to decide which is better. If this were really a meaningless question, you might as well flip a coin.)

Aiming for succinctness seems a good way to find new ideas. If you can do something that makes many different programs shorter, it is probably not a coincidence: you have probably discovered a useful new abstraction. You might even be able to write a program to help by searching source code for repeated patterns. Among other languages, those with a reputation for succinctness would be the ones to look to for new ideas: Forth, Joy, Icon.

Comparison

The first person to write about these issues, as far as I know, was Fred Brooks in the *Mythical Man Month*. He wrote that programmers seemed to generate about the same amount of code per day regardless of the language. When I first read this in my early twenties, it was a big surprise to me and seemed to have huge implications. It meant that (a) the only way to get software written faster was to use a more succinct language, and (b) someone who took the trouble to do this could leave competitors who didn't in the dust.

Brooks' hypothesis, if it's true, seems to be at the very heart of hacking. In the years since, I've paid close attention to any evidence I could get on the question, from formal studies to anecdotes about individual projects. I have seen nothing to contradict him.

I have not yet seen evidence that seemed to me conclusive, and I don't expect to. Studies like Lutz Prechelt's comparison of programming languages, while generating the kind of results I expected, tend to use problems that are too short to be meaningful tests. A better test of a language is what happens in programs that take a month to write. And the only real test, if you believe as I do that the main purpose of a language is to be good to think in (rather than just to tell a computer what to do once you've thought of it) is what new things you can write in it. So any language comparison where you have to meet a predefined spec is testing slightly the wrong thing.

The true test of a language is how well you can discover and solve new problems, not how well you can use it to solve a problem someone else has already formulated. These two are quite different criteria. In art, mediums like embroidery and mosaic work well if you know beforehand what you want to make, but are absolutely lousy if you don't. When you want to discover the image as you make it-- as you have to do with anything as complex as an image of a person, for example-- you need to use a more fluid medium like pencil or ink wash or oil paint. And indeed, the way tapestries and mosaics are made in practice is to make a

painting first, then copy it. (The word "cartoon" was originally used to describe a painting intended for this purpose).

What this means is that we are never likely to have accurate comparisons of the relative power of programming languages. We'll have precise comparisons, but not accurate ones. In particular, explicit studies for the purpose of comparing languages, because they will probably use small problems, and will necessarily use predefined problems, will tend to underestimate the power of the more powerful languages.

Reports from the field, though they will necessarily be less precise than "scientific" studies, are likely to be more meaningful. For example, Ulf Wiger of Ericsson did a study that concluded that Erlang was 4-10x more succinct than C++, and proportionately faster to develop software in:

> Comparisons between Ericsson-internal development projects indicate similar line/hour productivity, including all phases of software development, rather independently of which language (Erlang, PLEX, C, C++, or Java) was used. What differentiates the different languages then becomes source code volume.

The study also deals explictly with a point that was only implicit in Brooks' book (since he measured lines of debugged code): programs written in more powerful languages tend to have fewer bugs. That becomes an end in itself, possibly more important than programmer productivity, in applications like network switches.

The Taste Test

Ultimately, I think you have to go with your gut. What does it feel like to program in the language? I think the way to find (or design) the best language is to become hypersensitive to how well a language lets you think, then choose/design the language that feels best. If some language feature is awkward or restricting, don't worry, you'll know about it.

Such hypersensitivity will come at a cost. You'll find that you can't *stand* programming in clumsy languages. I find it unbearably restrictive to program in languages without macros, just as someone used to dynamic typing finds it unbearably restrictive to have to go back to programming in a language where you have to declare the type of every variable, and can't make a list of objects of different types.

I'm not the only one. I know many Lisp hackers that this has happened to. In fact, the most accurate measure of the relative power of programming languages might be the percentage of people who know the language who will take any job where they get to use that language, regardless of the application domain.

Restrictiveness

I think most hackers know what it means for a language to feel restrictive. What's

happening when you feel that? I think it's the same feeling you get when the street you want to take is blocked off, and you have to take a long detour to get where you wanted to go. There is something you want to say, and the language won't let you.

What's really going on here, I think, is that a restrictive language is one that isn't succinct enough. The problem is not simply that you can't say what you planned to. It's that the detour the language makes you take is *longer.* Try this thought experiment. Suppose there were some program you wanted to write, and the language wouldn't let you express it the way you planned to, but instead forced you to write the program in some other way that was *shorter.* For me at least, that wouldn't feel very restrictive. It would be like the street you wanted to take being blocked off, and the policeman at the intersection directing you to a shortcut instead of a detour. Great!

I think most (ninety percent?) of the feeling of restrictiveness comes from being forced to make the program you write in the language longer than one you have in your head. Restrictiveness is mostly lack of succinctness. So when a language feels restrictive, what that (mostly) means is that it isn't succinct enough, and when a language isn't succinct, it will feel restrictive.

Readability

The quote I began with mentions two other qualities, regularity and readability. I'm not sure what regularity is, or what advantage, if any, code that is regular and readable has over code that is merely readable. But I think I know what is meant by readability, and I think it is also related to succinctness.

We have to be careful here to distinguish between the readability of an individual line of code and the readability of the whole program. It's the second that matters. I agree that a line of Basic is likely to be more readable than a line of Lisp. But a program written in Basic is is going to have more lines than the same program written in Lisp (especially once you cross over into Greenspunland). The total effort of reading the Basic program will surely be greater.

> total effort = effort per line x number of lines

I'm not as sure that readability is directly proportionate to succinctness as I am that power is, but certainly succinctness is a factor (in the mathematical sense; see equation above) in readability. So it may not even be meaningful to say that the goal of a language is readability, not succinctness; it could be like saying the goal was readability, not readability.

What readability-per-line does mean, to the user encountering the language for the first time, is that source code will *look unthreatening*. So readability-per-line could be a good marketing decision, even if it is a bad design decision. It's isomorphic to the very successful technique of letting people pay in installments: instead of frightening them with a high upfront price, you tell them the low monthly payment.

Installment plans are a net lose for the buyer, though, as mere readability-per-line probably is for the programmer. The buyer is going to make a *lot* of those low, low payments; and the programmer is going to read a *lot* of those individually readable lines.

This tradeoff predates programming languages. If you're used to reading novels and newspaper articles, your first experience of reading a math paper can be dismaying. It could take half an hour to read a single page. And yet, I am pretty sure that the notation is not the problem, even though it may feel like it is. The math paper is hard to read because the ideas are hard. If you expressed the same ideas in prose (as mathematicians had to do before they evolved succinct notations), they wouldn't be any easier to read, because the paper would grow to the size of a book.

To What Extent?

A number of people have rejected the idea that succinctness = power. I think it would be more useful, instead of simply arguing that they are the same or aren't, to ask: to what *extent* does succinctness = power? Because clearly succinctness is a large part of what higher-level languages are for. If it is not all they're for, then what else are they for, and how important, relatively, are these other functions?

I'm not proposing this just to make the debate more civilized. I really want to know the answer. When, if ever, is a language too succinct for its own good?

The hypothesis I began with was that, except in pathological examples, I thought succinctness could be considered identical with power. What I meant was that in any language anyone would design, they would be identical, but that if someone wanted to design a language explicitly to disprove this hypothesis, they could probably do it. I'm not even sure of that, actually.

Languages, not Programs

We should be clear that we are talking about the succinctness of languages, not of individual programs. It certainly is possible for individual programs to be written too densely.

I wrote about this in <u>On Lisp</u>. A complex macro may have to save many times its own length to be justified. If writing some hairy macro could save you ten lines of code every time you use it, and the macro is itself ten lines of code, then you get a net saving in lines if you use it more than once. But that could still be a bad move, because macro definitions are harder to read than ordinary code. You might have to use the macro ten or twenty times before it yielded a net improvement in readability.

I'm sure every language has such tradeoffs (though I suspect the stakes get higher as the language gets more powerful). Every programmer must have seen code that some clever person has made marginally shorter by using dubious programming

tricks.

So there is no argument about that-- at least, not from me. Individual programs can certainly be too succinct for their own good. The question is, can a language be? Can a language compel programmers to write code that's short (in elements) at the expense of overall readability?

One reason it's hard to imagine a language being too succinct is that if there were some excessively compact way to phrase something, there would probably also be a longer way. For example, if you felt Lisp programs using a lot of macros or higher-order functions were too dense, you could, if you preferred, write code that was isomorphic to Pascal. If you don't want to express factorial in Arc as a call to a higher-order function

```
(rec zero 1 * 1-)
```

you can also write out a recursive definition:

```
(rfn fact (x) (if (zero x) 1 (* x (fact (1- x)))))
```

Though I can't off the top of my head think of any examples, I am interested in the question of whether a language could be too succinct. Are there languages that force you to write code in a way that is crabbed and incomprehensible? If anyone has examples, I would be very interested to see them.

(Reminder: What I'm looking for are programs that are very dense according to the metric of "elements" sketched above, not merely programs that are short because delimiters can be omitted and everything has a one-character name.)

■

Japanese Translation

■

Russian Translation

■

J

K

Revenge of the Nerds

May 2002

"We were after the C++ programmers. We managed to drag a lot of them about halfway to Lisp."

- Guy Steele, co-author of the Java spec

In the software business there is an ongoing struggle between the pointy-headed academics, and another equally formidable force, the pointy-haired bosses. Everyone knows who the pointy-haired boss is, right? I think most people in the technology world not only recognize this cartoon character, but know the actual person in their company that he is modelled upon.

The pointy-haired boss miraculously combines two qualities that are common by themselves, but rarely seen together: (a) he knows nothing whatsoever about technology, and (b) he has very strong opinions about it.

Suppose, for example, you need to write a piece of software. The pointy-haired boss has no idea how this software has to work, and can't tell one programming language from another, and yet he knows what language you should write it in. Exactly. He thinks you should write it in Java.

Why does he think this? Let's take a look inside the brain of the pointy-haired boss. What he's thinking is something like this. Java is a standard. I know it must be, because I read about it in the press all the time. Since it is a standard, I won't get in trouble for using it. And that also means there will always be lots of Java programmers, so if the programmers working for me now quit, as programmers working for me mysteriously always do, I can easily replace them.

Well, this doesn't sound that unreasonable. But it's all based on one unspoken assumption, and that assumption turns out to be false. The pointy-haired boss believes that all programming languages are pretty much equivalent. If that were true, he would be right on target. If languages are all equivalent, sure, use whatever language everyone else is using.

But all languages are not equivalent, and I think I can prove this to you without

even getting into the differences between them. If you asked the pointy-haired boss in 1992 what language software should be written in, he would have answered with as little hesitation as he does today. Software should be written in C++. But if languages are all equivalent, why should the pointy-haired boss's opinion ever change? In fact, why should the developers of Java have even bothered to create a new language?

Presumably, if you create a new language, it's because you think it's better in some way than what people already had. And in fact, Gosling makes it clear in the first Java white paper that Java was designed to fix some problems with C++. So there you have it: languages are not all equivalent. If you follow the trail through the pointy-haired boss's brain to Java and then back through Java's history to its origins, you end up holding an idea that contradicts the assumption you started with.

So, who's right? James Gosling, or the pointy-haired boss? Not surprisingly, Gosling is right. Some languages *are* better, for certain problems, than others. And you know, that raises some interesting questions. Java was designed to be better, for certain problems, than C++. What problems? When is Java better and when is C++? Are there situations where other languages are better than either of them?

Once you start considering this question, you have opened a real can of worms. If the pointy-haired boss had to think about the problem in its full complexity, it would make his brain explode. As long as he considers all languages equivalent, all he has to do is choose the one that seems to have the most momentum, and since that is more a question of fashion than technology, even he can probably get the right answer. But if languages vary, he suddenly has to solve two simultaneous equations, trying to find an optimal balance between two things he knows nothing about: the relative suitability of the twenty or so leading languages for the problem he needs to solve, and the odds of finding programmers, libraries, etc. for each. If that's what's on the other side of the door, it is no surprise that the pointy-haired boss doesn't want to open it.

The disadvantage of believing that all programming languages are equivalent is that it's not true. But the advantage is that it makes your life a lot simpler. And I think that's the main reason the idea is so widespread. It is a *comfortable* idea.

We know that Java must be pretty good, because it is the cool, new programming language. Or is it? If you look at the world of programming languages from a distance, it looks like Java is the latest thing. (From far enough away, all you can see is the large, flashing billboard paid for by Sun.) But if you look at this world up close, you find that there are degrees of coolness. Within the hacker subculture, there is another language called Perl that is considered a lot cooler than Java. Slashdot, for example, is generated by Perl. I don't think you would find those guys using Java Server Pages. But there is another, newer language, called Python, whose users tend to look down on Perl, and [more](#) waiting in the wings.

If you look at these languages in order, Java, Perl, Python, you notice an

interesting pattern. At least, you notice this pattern if you are a Lisp hacker. Each one is progressively more like Lisp. Python copies even features that many Lisp hackers consider to be mistakes. You could translate simple Lisp programs into Python line for line. It's 2002, and programming languages have almost caught up with 1958.

Catching Up with Math

What I mean is that Lisp was first discovered by John McCarthy in 1958, and popular programming languages are only now catching up with the ideas he developed then.

Now, how could that be true? Isn't computer technology something that changes very rapidly? I mean, in 1958, computers were refrigerator-sized behemoths with the processing power of a wristwatch. How could any technology that old even be relevant, let alone superior to the latest developments?

I'll tell you how. It's because Lisp was not really designed to be a programming language, at least not in the sense we mean today. What we mean by a programming language is something we use to tell a computer what to do. McCarthy did eventually intend to develop a programming language in this sense, but the Lisp that we actually ended up with was based on something separate that he did as a theoretical exercise-- an effort to define a more convenient alternative to the Turing Machine. As McCarthy said later,

> Another way to show that Lisp was neater than Turing machines was to write a universal Lisp function and show that it is briefer and more comprehensible than the description of a universal Turing machine. This was the Lisp function *eval*..., which computes the value of a Lisp expression.... Writing *eval* required inventing a notation representing Lisp functions as Lisp data, and such a notation was devised for the purposes of the paper with no thought that it would be used to express Lisp programs in practice.

What happened next was that, some time in late 1958, Steve Russell, one of McCarthy's grad students, looked at this definition of *eval* and realized that if he translated it into machine language, the result would be a Lisp interpreter.

This was a big surprise at the time. Here is what McCarthy said about it later in an interview:

> Steve Russell said, look, why don't I program this *eval*..., and I said to him, ho, ho, you're confusing theory with practice, this *eval* is intended for reading, not for computing. But he went ahead and did it. That is, he compiled the *eval* in my paper into [IBM] 704 machine code, fixing bugs, and then advertised this as a Lisp interpreter, which it certainly was. So at that point Lisp had essentially the form that it has today....

Suddenly, in a matter of weeks I think, McCarthy found his theoretical exercise transformed into an actual programming language-- and a more powerful one than

he had intended.

So the short explanation of why this 1950s language is not obsolete is that it was not technology but math, and math doesn't get stale. The right thing to compare Lisp to is not 1950s hardware, but, say, the Quicksort algorithm, which was discovered in 1960 and is still the fastest general-purpose sort.

There is one other language still surviving from the 1950s, Fortran, and it represents the opposite approach to language design. Lisp was a piece of theory that unexpectedly got turned into a programming language. Fortran was developed intentionally as a programming language, but what we would now consider a very low-level one.

Fortran I, the language that was developed in 1956, was a very different animal from present-day Fortran. Fortran I was pretty much assembly language with math. In some ways it was less powerful than more recent assembly languages; there were no subroutines, for example, only branches. Present-day Fortran is now arguably closer to Lisp than to Fortran I.

Lisp and Fortran were the trunks of two separate evolutionary trees, one rooted in math and one rooted in machine architecture. These two trees have been converging ever since. Lisp started out powerful, and over the next twenty years got fast. So-called mainstream languages started out fast, and over the next forty years gradually got more powerful, until now the most advanced of them are fairly close to Lisp. Close, but they are still missing a few things....

What Made Lisp Different

When it was first developed, Lisp embodied nine new ideas. Some of these we now take for granted, others are only seen in more advanced languages, and two are still unique to Lisp. The nine ideas are, in order of their adoption by the mainstream,

1. Conditionals. A conditional is an if-then-else construct. We take these for granted now, but Fortran I didn't have them. It had only a conditional goto closely based on the underlying machine instruction.

2. A function type. In Lisp, functions are a data type just like integers or strings. They have a literal representation, can be stored in variables, can be passed as arguments, and so on.

3. Recursion. Lisp was the first programming language to support it.

4. Dynamic typing. In Lisp, all variables are effectively pointers. Values are what have types, not variables, and assigning or binding variables means copying pointers, not what they point to.

5. Garbage-collection.

6. Programs composed of expressions. Lisp programs are trees of expressions, each of which returns a value. This is in contrast to Fortran and most succeeding languages, which distinguish between expressions and statements.

It was natural to have this distinction in Fortran I because you could not nest statements. And so while you needed expressions for math to work, there was no point in making anything else return a value, because there could not be anything waiting for it.

This limitation went away with the arrival of block-structured languages, but by then it was too late. The distinction between expressions and statements was entrenched. It spread from Fortran into Algol and then to both their descendants.

7. A symbol type. Symbols are effectively pointers to strings stored in a hash table. So you can test equality by comparing a pointer, instead of comparing each character.

8. A notation for code using trees of symbols and constants.

9. The whole language there all the time. There is no real distinction between read-time, compile-time, and runtime. You can compile or run code while reading, read or run code while compiling, and read or compile code at runtime.

Running code at read-time lets users reprogram Lisp's syntax; running code at compile-time is the basis of macros; compiling at runtime is the basis of Lisp's use as an extension language in programs like Emacs; and reading at runtime enables programs to communicate using s-expressions, an idea recently reinvented as XML.

When Lisp first appeared, these ideas were far removed from ordinary programming practice, which was dictated largely by the hardware available in the late 1950s. Over time, the default language, embodied in a succession of popular languages, has gradually evolved toward Lisp. Ideas 1-5 are now widespread. Number 6 is starting to appear in the mainstream. Python has a form of 7, though there doesn't seem to be any syntax for it.

As for number 8, this may be the most interesting of the lot. Ideas 8 and 9 only became part of Lisp by accident, because Steve Russell implemented something McCarthy had never intended to be implemented. And yet these ideas turn out to be responsible for both Lisp's strange appearance and its most distinctive features. Lisp looks strange not so much because it has a strange syntax as because it has no syntax; you express programs directly in the parse trees that get built behind the scenes when other languages are parsed, and these trees are made of lists, which are Lisp data structures.

Expressing the language in its own data structures turns out to be a very powerful feature. Ideas 8 and 9 together mean that you can write programs that write programs. That may sound like a bizarre idea, but it's an everyday thing in Lisp. The most common way to do it is with something called a *macro.*

The term "macro" does not mean in Lisp what it means in other languages. A Lisp macro can be anything from an abbreviation to a compiler for a new language. If you want to really understand Lisp, or just expand your programming horizons, I would learn more about macros.

Macros (in the Lisp sense) are still, as far as I know, unique to Lisp. This is partly because in order to have macros you probably have to make your language look as strange as Lisp. It may also be because if you do add that final increment of power, you can no longer claim to have invented a new language, but only a new dialect of Lisp.

I mention this mostly as a joke, but it is quite true. If you define a language that has car, cdr, cons, quote, cond, atom, eq, and a notation for functions expressed as lists, then you can build all the rest of Lisp out of it. That is in fact the defining quality of Lisp: it was in order to make this so that McCarthy gave Lisp the shape it has.

Where Languages Matter

So suppose Lisp does represent a kind of limit that mainstream languages are approaching asymptotically-- does that mean you should actually use it to write software? How much do you lose by using a less powerful language? Isn't it wiser, sometimes, not to be at the very edge of innovation? And isn't popularity to some extent its own justification? Isn't the pointy-haired boss right, for example, to want to use a language for which he can easily hire programmers?

There are, of course, projects where the choice of programming language doesn't matter much. As a rule, the more demanding the application, the more leverage you get from using a powerful language. But plenty of projects are not demanding at all. Most programming probably consists of writing little glue programs, and for little glue programs you can use any language that you're already familiar with and that has good libraries for whatever you need to do. If you just need to feed data from one Windows app to another, sure, use Visual Basic.

You can write little glue programs in Lisp too (I use it as a desktop calculator), but the biggest win for languages like Lisp is at the other end of the spectrum, where you need to write sophisticated programs to solve hard problems in the face of fierce competition. A good example is the airline fare search program that ITA Software licenses to Orbitz. These guys entered a market already dominated by two big, entrenched competitors, Travelocity and Expedia, and seem to have just humiliated them technologically.

The core of ITA's application is a 200,000 line Common Lisp program that searches many orders of magnitude more possibilities than their competitors, who apparently are still using mainframe-era programming techniques. (Though ITA is also in a sense using a mainframe-era programming language.) I have never seen any of ITA's code, but according to one of their top hackers they use a lot of

macros, and I am not surprised to hear it.

Centripetal Forces

I'm not saying there is no cost to using uncommon technologies. The pointy-haired boss is not completely mistaken to worry about this. But because he doesn't understand the risks, he tends to magnify them.

I can think of three problems that could arise from using less common languages. Your programs might not work well with programs written in other languages. You might have fewer libraries at your disposal. And you might have trouble hiring programmers.

How much of a problem is each of these? The importance of the first varies depending on whether you have control over the whole system. If you're writing software that has to run on a remote user's machine on top of a buggy, closed operating system (I mention no names), there may be advantages to writing your application in the same language as the OS. But if you control the whole system and have the source code of all the parts, as ITA presumably does, you can use whatever languages you want. If any incompatibility arises, you can fix it yourself.

In server-based applications you can get away with using the most advanced technologies, and I think this is the main cause of what Jonathan Erickson calls the "programming language renaissance." This is why we even hear about new languages like Perl and Python. We're not hearing about these languages because people are using them to write Windows apps, but because people are using them on servers. And as software shifts off the desktop and onto servers (a future even Microsoft seems resigned to), there will be less and less pressure to use middle-of-the-road technologies.

As for libraries, their importance also depends on the application. For less demanding problems, the availability of libraries can outweigh the intrinsic power of the language. Where is the breakeven point? Hard to say exactly, but wherever it is, it is short of anything you'd be likely to call an application. If a company considers itself to be in the software business, and they're writing an application that will be one of their products, then it will probably involve several hackers and take at least six months to write. In a project of that size, powerful languages probably start to outweigh the convenience of pre-existing libraries.

The third worry of the pointy-haired boss, the difficulty of hiring programmers, I think is a red herring. How many hackers do you need to hire, after all? Surely by now we all know that software is best developed by teams of less than ten people. And you shouldn't have trouble hiring hackers on that scale for any language anyone has ever heard of. If you can't find ten Lisp hackers, then your company is probably based in the wrong city for developing software.

In fact, choosing a more powerful language probably decreases the size of the team you need, because (a) if you use a more powerful language you probably

won't need as many hackers, and (b) hackers who work in more advanced languages are likely to be smarter.

I'm not saying that you won't get a lot of pressure to use what are perceived as "standard" technologies. At Viaweb (now Yahoo Store), we raised some eyebrows among VCs and potential acquirers by using Lisp. But we also raised eyebrows by using generic Intel boxes as servers instead of "industrial strength" servers like Suns, for using a then-obscure open-source Unix variant called FreeBSD instead of a real commercial OS like Windows NT, for ignoring a supposed e-commerce standard called SET that no one now even remembers, and so on.

You can't let the suits make technical decisions for you. Did it alarm some potential acquirers that we used Lisp? Some, slightly, but if we hadn't used Lisp, we wouldn't have been able to write the software that made them want to buy us. What seemed like an anomaly to them was in fact cause and effect.

If you start a startup, don't design your product to please VCs or potential acquirers. *Design your product to please the users.* If you win the users, everything else will follow. And if you don't, no one will care how comfortingly orthodox your technology choices were.

The Cost of Being Average

How much do you lose by using a less powerful language? There is actually some data out there about that.

The most convenient measure of power is probably code size. The point of high-level languages is to give you bigger abstractions-- bigger bricks, as it were, so you don't need as many to build a wall of a given size. So the more powerful the language, the shorter the program (not simply in characters, of course, but in distinct elements).

How does a more powerful language enable you to write shorter programs? One technique you can use, if the language will let you, is something called bottom-up programming. Instead of simply writing your application in the base language, you build on top of the base language a language for writing programs like yours, then write your program in it. The combined code can be much shorter than if you had written your whole program in the base language-- indeed, this is how most compression algorithms work. A bottom-up program should be easier to modify as well, because in many cases the language layer won't have to change at all.

Code size is important, because the time it takes to write a program depends mostly on its length. If your program would be three times as long in another language, it will take three times as long to write-- and you can't get around this by hiring more people, because beyond a certain size new hires are actually a net lose. Fred Brooks described this phenomenon in his famous book *The Mythical Man-Month,* and everything I've seen has tended to confirm what he said.

So how much shorter are your programs if you write them in Lisp? Most of the numbers I've heard for Lisp versus C, for example, have been around 7-10x. But a recent article about ITA in *New Architect* magazine said that "one line of Lisp can replace 20 lines of C," and since this article was full of quotes from ITA's president, I assume they got this number from ITA. If so then we can put some faith in it; ITA's software includes a lot of C and C++ as well as Lisp, so they are speaking from experience.

My guess is that these multiples aren't even constant. I think they increase when you face harder problems and also when you have smarter programmers. A really good hacker can squeeze more out of better tools.

As one data point on the curve, at any rate, if you were to compete with ITA and chose to write your software in C, they would be able to develop software twenty times faster than you. If you spent a year on a new feature, they'd be able to duplicate it in less than three weeks. Whereas if they spent just three months developing something new, it would be *five years* before you had it too.

And you know what? That's the best-case scenario. When you talk about code-size ratios, you're implicitly assuming that you can actually write the program in the weaker language. But in fact there are limits on what programmers can do. If you're trying to solve a hard problem with a language that's too low-level, you reach a point where there is just too much to keep in your head at once.

So when I say it would take ITA's imaginary competitor five years to duplicate something ITA could write in Lisp in three months, I mean five years if nothing goes wrong. In fact, the way things work in most companies, any development project that would take five years is likely never to get finished at all.

I admit this is an extreme case. ITA's hackers seem to be unusually smart, and C is a pretty low-level language. But in a competitive market, even a differential of two or three to one would be enough to guarantee that you'd always be behind.

A Recipe

This is the kind of possibility that the pointy-haired boss doesn't even want to think about. And so most of them don't. Because, you know, when it comes down to it, the pointy-haired boss doesn't mind if his company gets their ass kicked, so long as no one can prove it's his fault. The safest plan for him personally is to stick close to the center of the herd.

Within large organizations, the phrase used to describe this approach is "industry best practice." Its purpose is to shield the pointy-haired boss from responsibility: if he chooses something that is "industry best practice," and the company loses, he can't be blamed. He didn't choose, the industry did.

I believe this term was originally used to describe accounting methods and so on. What it means, roughly, is *don't do anything weird.* And in accounting that's

probably a good idea. The terms "cutting-edge" and "accounting" do not sound good together. But when you import this criterion into decisions about technology, you start to get the wrong answers.

Technology often *should* be cutting-edge. In programming languages, as Erann Gat has pointed out, what "industry best practice" actually gets you is not the best, but merely the average. When a decision causes you to develop software at a fraction of the rate of more aggressive competitors, "best practice" is a misnomer.

So here we have two pieces of information that I think are very valuable. In fact, I know it from my own experience. Number 1, languages vary in power. Number 2, most managers deliberately ignore this. Between them, these two facts are literally a recipe for making money. ITA is an example of this recipe in action. If you want to win in a software business, just take on the hardest problem you can find, use the most powerful language you can get, and wait for your competitors' pointy-haired bosses to revert to the mean.

Appendix: Power

As an illustration of what I mean about the relative power of programming languages, consider the following problem. We want to write a function that generates accumulators-- a function that takes a number n, and returns a function that takes another number i and returns n incremented by i.

(That's *incremented by*, not plus. An accumulator has to accumulate.)

In Common Lisp this would be

```
(defun foo (n)
  (lambda (i) (incf n i)))
```

and in Perl 5,

```
sub foo {
  my ($n) = @_;
  sub {$n += shift}
}
```

which has more elements than the Lisp version because you have to extract parameters manually in Perl.

In Smalltalk the code is slightly longer than in Lisp

```
foo: n
  |s|
```

```
    s := n.
    ^[:i| s := s+i. ]
```
because although in general lexical variables work, you can't do an assignment to a parameter, so you have to create a new variable s.

In Javascript the example is, again, slightly longer, because Javascript retains the distinction between statements and expressions, so you need explicit `return` statements to return values:

```
function foo(n) {
  return function (i) {
           return n += i } }
```

(To be fair, Perl also retains this distinction, but deals with it in typical Perl fashion by letting you omit `return`s.)

If you try to translate the Lisp/Perl/Smalltalk/Javascript code into Python you run into some limitations. Because Python doesn't fully support lexical variables, you have to create a data structure to hold the value of n. And although Python does have a function data type, there is no literal representation for one (unless the body is only a single expression) so you need to create a named function to return. This is what you end up with:

```
def foo(n):
  s = [n]
  def bar(i):
    s[0] += i
    return s[0]
  return bar
```

Python users might legitimately ask why they can't just write

```
def foo(n):
  return lambda i: return n += i
```

or even

```
def foo(n):
  lambda i: n += i
```

and my guess is that they probably will, one day. (But if they don't want to wait for Python to evolve the rest of the way into Lisp, they could always just...)

In OO languages, you can, to a limited extent, simulate a closure (a function that refers to variables defined in enclosing scopes) by defining a class with one method and a field to replace each variable from an enclosing scope. This makes the programmer do the kind of code analysis that would be done by the compiler in a language with full support for lexical scope, and it won't work if more than one function refers to the same variable, but it is enough in simple cases like this.

Python experts seem to agree that this is the preferred way to solve the problem in Python, writing either

```
def foo(n):
  class acc:
    def __init__(self, s):
        self.s = s
    def inc(self, i):
        self.s += i
        return self.s
  return acc(n).inc
```

or

```
class foo:
  def __init__(self, n):
      self.n = n
  def __call__(self, i):
      self.n += i
      return self.n
```

I include these because I wouldn't want Python advocates to say I was misrepresenting the language, but both seem to me more complex than the first version. You're doing the same thing, setting up a separate place to hold the accumulator; it's just a field in an object instead of the head of a list. And the use of these special, reserved field names, especially __call__, seems a bit of a hack.

In the rivalry between Perl and Python, the claim of the Python hackers seems to be that that Python is a more elegant alternative to Perl, but what this case shows is that power is the ultimate elegance: the Perl program is simpler (has fewer elements), even if the syntax is a bit uglier.

How about other languages? In the other languages mentioned in this talk-- Fortran, C, C++, Java, and Visual Basic-- it is not clear whether you can actually solve this problem. Ken Anderson says that the following code is about as close as you can get in Java:

```
public interface Inttoint {
  public int call(int i);
}
```

```
public static Inttoint foo(final int n) {
  return new Inttoint() {
    int s = n;
    public int call(int i) {
    s = s + i;
    return s;
    }};
}
```

This falls short of the spec because it only works for integers. After many email exchanges with Java hackers, I would say that writing a properly polymorphic

version that behaves like the preceding examples is somewhere between damned awkward and impossible. If anyone wants to write one I'd be very curious to see it, but I personally have timed out.

It's not literally true that you can't solve this problem in other languages, of course. The fact that all these languages are Turing-equivalent means that, strictly speaking, you can write any program in any of them. So how would you do it? In the limit case, by writing a Lisp interpreter in the less powerful language.

That sounds like a joke, but it happens so often to varying degrees in large programming projects that there is a name for the phenomenon, Greenspun's Tenth Rule:

> Any sufficiently complicated C or Fortran program contains an ad hoc informally-specified bug-ridden slow implementation of half of Common Lisp.

If you try to solve a hard problem, the question is not whether you will use a powerful enough language, but whether you will (a) use a powerful language, (b) write a de facto interpreter for one, or (c) yourself become a human compiler for one. We see this already begining to happen in the Python example, where we are in effect simulating the code that a compiler would generate to implement a lexical variable.

This practice is not only common, but institutionalized. For example, in the OO world you hear a good deal about "patterns". I wonder if these patterns are not sometimes evidence of case (c), the human compiler, at work. When I see patterns in my programs, I consider it a sign of trouble. The shape of a program should reflect only the problem it needs to solve. Any other regularity in the code is a sign, to me at least, that I'm using abstractions that aren't powerful enough-- often that I'm generating by hand the expansions of some macro that I need to write.

Notes

- The IBM 704 CPU was about the size of a refrigerator, but a lot heavier. The CPU weighed 3150 pounds, and the 4K of RAM was in a separate box weighing another 4000 pounds. The Sub-Zero 690, one of the largest household refrigerators, weighs 656 pounds.

- Steve Russell also wrote the first (digital) computer game, Spacewar, in 1962.

- If you want to trick a pointy-haired boss into letting you write software in Lisp, you could try telling him it's XML.

- Here is the accumulator generator in other Lisp dialects:

```
Scheme: (define (foo n)
          (lambda (i) (set! n (+ n i)) n))
Goo:    (df foo (n) (op incf n _)))
Arc:    (def foo (n) [++ n _])
```

- Erann Gat's sad tale about "industry best practice" at JPL inspired me to address this generally misapplied phrase.

- Peter Norvig found that 16 of the 23 patterns in *Design Patterns* were "invisible or simpler" in Lisp.

- Thanks to the many people who answered my questions about various languages and/or read drafts of this, including Ken Anderson, Trevor Blackwell, Erann Gat, Dan Giffin, Sarah Harlin, Jeremy Hylton, Robert Morris, Peter Norvig, Guy Steele, and Anton van Straaten. They bear no blame for any opinions expressed.

Related:

Many people have responded to this talk, so I have set up an additional page to deal with the issues they have raised: Re: Revenge of the Nerds.

It also set off an extensive and often useful discussion on the LL1 mailing list. See particularly the mail by Anton van Straaten on semantic compression.

Some of the mail on LL1 led me to try to go deeper into the subject of language power in Succinctness is Power.

A larger set of canonical implementations of the accumulator generator benchmark are collected together on their own page.

Japanese Translation, Spanish Translation, Chinese Translation

You'll find this essay and 14 others in ***Hackers & Painters***.

A Plan for Spam

Like to build things? Try [Hacker News](#).

August 2002

(This article describes the spam-filtering techniques used in the spamproof web-based mail reader we built to exercise [Arc](#). An improved algorithm is described in [Better Bayesian Filtering](#).)

I think it's possible to stop spam, and that content-based filters are the way to do it. The Achilles heel of the spammers is their message. They can circumvent any other barrier you set up. They have so far, at least. But they have to deliver their message, whatever it is. If we can write software that recognizes their messages, there is no way they can get around that.

— — —

To the recipient, spam is easily recognizable. If you hired someone to read your mail and discard the spam, they would have little trouble doing it. How much do we have to do, short of AI, to automate this process?

I think we will be able to solve the problem with fairly simple algorithms. In fact, I've found that you can filter present-day spam acceptably well using nothing more than a Bayesian combination of the spam probabilities of individual words. Using a slightly tweaked (as described below) Bayesian filter, we now miss less than 5 per 1000 spams, with 0 false positives.

The statistical approach is not usually the first one people try when they write spam filters. Most hackers' first instinct is to try to write software that recognizes individual properties of spam. You look at spams and you think, the gall of these guys to try sending me mail that begins "Dear Friend" or has a subject line that's all uppercase and ends in eight exclamation points. I can filter out that stuff with about one line of code.

And so you do, and in the beginning it works. A few simple rules will take a big bite out of your incoming spam. Merely looking for the word "click" will catch 79.7% of the emails in my spam corpus, with only 1.2% false positives.

I spent about six months writing software that looked for individual spam features

before I tried the statistical approach. What I found was that recognizing that last few percent of spams got very hard, and that as I made the filters stricter I got more false positives.

False positives are innocent emails that get mistakenly identified as spams. For most users, missing legitimate email is an order of magnitude worse than receiving spam, so a filter that yields false positives is like an acne cure that carries a risk of death to the patient.

The more spam a user gets, the less likely he'll be to notice one innocent mail sitting in his spam folder. And strangely enough, the better your spam filters get, the more dangerous false positives become, because when the filters are really good, users will be more likely to ignore everything they catch.

I don't know why I avoided trying the statistical approach for so long. I think it was because I got addicted to trying to identify spam features myself, as if I were playing some kind of competitive game with the spammers. (Nonhackers don't often realize this, but most hackers are very competitive.) When I did try statistical analysis, I found immediately that it was much cleverer than I had been. It discovered, of course, that terms like "virtumundo" and "teens" were good indicators of spam. But it also discovered that "per" and "FL" and "ff0000" are good indicators of spam. In fact, "ff0000" (html for bright red) turns out to be as good an indicator of spam as any pornographic term.

— — —

Here's a sketch of how I do statistical filtering. I start with one corpus of spam and one of nonspam mail. At the moment each one has about 4000 messages in it. I scan the entire text, including headers and embedded html and javascript, of each message in each corpus. I currently consider alphanumeric characters, dashes, apostrophes, and dollar signs to be part of tokens, and everything else to be a token separator. (There is probably room for improvement here.) I ignore tokens that are all digits, and I also ignore html comments, not even considering them as token separators.

I count the number of times each token (ignoring case, currently) occurs in each corpus. At this stage I end up with two large hash tables, one for each corpus, mapping tokens to number of occurrences.

Next I create a third hash table, this time mapping each token to the probability that an email containing it is a spam, which I calculate as follows [1]:

```
(let ((g (* 2 (or (gethash word good) 0)))
      (b (or (gethash word bad) 0)))
   (unless (< (+ g b) 5)
     (max .01
          (min .99 (float (/ (min 1 (/ b nbad))
```

```
                    (+ (min 1 (/ g ngood))
                       (min 1 (/ b nbad))))))))))
```

where `word` is the token whose probability we're calculating, `good` and `bad` are the hash tables I created in the first step, and `ngood` and `nbad` are the number of nonspam and spam messages respectively.

I explained this as code to show a couple of important details. I want to bias the probabilities slightly to avoid false positives, and by trial and error I've found that a good way to do it is to double all the numbers in `good`. This helps to distinguish between words that occasionally do occur in legitimate email and words that almost never do. I only consider words that occur more than five times in total (actually, because of the doubling, occurring three times in nonspam mail would be enough). And then there is the question of what probability to assign to words that occur in one corpus but not the other. Again by trial and error I chose .01 and .99. There may be room for tuning here, but as the corpus grows such tuning will happen automatically anyway.

The especially observant will notice that while I consider each corpus to be a single long stream of text for purposes of counting occurrences, I use the number of emails in each, rather than their combined length, as the divisor in calculating spam probabilities. This adds another slight bias to protect against false positives.

When new mail arrives, it is scanned into tokens, and the most interesting fifteen tokens, where interesting is measured by how far their spam probability is from a neutral .5, are used to calculate the probability that the mail is spam. If `probs` is a list of the fifteen individual probabilities, you calculate the [combined] probability thus:

```
(let ((prod (apply #'* probs)))
  (/ prod (+ prod (apply #'* (mapcar #'(lambda (x)
                                         (- 1 x))
                                     probs)))))
```

One question that arises in practice is what probability to assign to a word you've never seen, i.e. one that doesn't occur in the hash table of word probabilities. I've found, again by trial and error, that .4 is a good number to use. If you've never seen a word before, it is probably fairly innocent; spam words tend to be all too familiar.

There are examples of this algorithm being applied to actual emails in an appendix at the end.

I treat mail as spam if the algorithm above gives it a probability of more than .9 of being spam. But in practice it would not matter much where I put this threshold, because few probabilities end up in the middle of the range.

— — —

One great advantage of the statistical approach is that you don't have to read so many spams. Over the past six months, I've read literally thousands of spams, and it is really kind of demoralizing. Norbert Wiener said if you compete with slaves you become a slave, and there is something similarly degrading about competing with spammers. To recognize individual spam features you have to try to get into the mind of the spammer, and frankly I want to spend as little time inside the minds of spammers as possible.

But the real advantage of the Bayesian approach, of course, is that you know what you're measuring. Feature-recognizing filters like SpamAssassin assign a spam "score" to email. The Bayesian approach assigns an actual probability. The problem with a "score" is that no one knows what it means. The user doesn't know what it means, but worse still, neither does the developer of the filter. How many *points* should an email get for having the word "sex" in it? A probability can of course be mistaken, but there is little ambiguity about what it means, or how evidence should be combined to calculate it. Based on my corpus, "sex" indicates a .97 probability of the containing email being a spam, whereas "sexy" indicates .99 probability. And Bayes' Rule, equally unambiguous, says that an email containing both words would, in the (unlikely) absence of any other evidence, have a 99.97% chance of being a spam.

Because it is measuring probabilities, the Bayesian approach considers all the evidence in the email, both good and bad. Words that occur disproportionately *rarely* in spam (like "though" or "tonight" or "apparently") contribute as much to decreasing the probability as bad words like "unsubscribe" and "opt-in" do to increasing it. So an otherwise innocent email that happens to include the word "sex" is not going to get tagged as spam.

Ideally, of course, the probabilities should be calculated individually for each user. I get a lot of email containing the word "Lisp", and (so far) no spam that does. So a word like that is effectively a kind of password for sending mail to me. In my earlier spam-filtering software, the user could set up a list of such words and mail containing them would automatically get past the filters. On my list I put words like "Lisp" and also my zipcode, so that (otherwise rather spammy-sounding) receipts from online orders would get through. I thought I was being very clever, but I found that the Bayesian filter did the same thing for me, and moreover discovered of a lot of words I hadn't thought of.

When I said at the start that our filters let through less than 5 spams per 1000 with 0 false positives, I'm talking about filtering my mail based on a corpus of my mail. But these numbers are not misleading, because that is the approach I'm advocating: filter each user's mail based on the spam and nonspam mail he receives. Essentially, each user should have two delete buttons, ordinary delete and delete-as-spam. Anything deleted as spam goes into the spam corpus, and everything else goes into the nonspam corpus.

You could start users with a seed filter, but ultimately each user should have his own per-word probabilities based on the actual mail he receives. This (a) makes

the filters more effective, (b) lets each user decide their own precise definition of spam, and (c) perhaps best of all makes it hard for spammers to tune mails to get through the filters. If a lot of the brain of the filter is in the individual databases, then merely tuning spams to get through the seed filters won't guarantee anything about how well they'll get through individual users' varying and much more trained filters.

Content-based spam filtering is often combined with a whitelist, a list of senders whose mail can be accepted with no filtering. One easy way to build such a whitelist is to keep a list of every address the user has ever sent mail to. If a mail reader has a delete-as-spam button then you could also add the from address of every email the user has deleted as ordinary trash.

I'm an advocate of whitelists, but more as a way to save computation than as a way to improve filtering. I used to think that whitelists would make filtering easier, because you'd only have to filter email from people you'd never heard from, and someone sending you mail for the first time is constrained by convention in what they can say to you. Someone you already know might send you an email talking about sex, but someone sending you mail for the first time would not be likely to. The problem is, people can have more than one email address, so a new from-address doesn't guarantee that the sender is writing to you for the first time. It is not unusual for an old friend (especially if he is a hacker) to suddenly send you an email with a new from-address, so you can't risk false positives by filtering mail from unknown addresses especially stringently.

In a sense, though, my filters do themselves embody a kind of whitelist (and blacklist) because they are based on entire messages, including the headers. So to that extent they "know" the email addresses of trusted senders and even the routes by which mail gets from them to me. And they know the same about spam, including the server names, mailer versions, and protocols.

— — —

If I thought that I could keep up current rates of spam filtering, I would consider this problem solved. But it doesn't mean much to be able to filter out most present-day spam, because spam evolves. Indeed, most antispam techniques so far have been like pesticides that do nothing more than create a new, resistant strain of bugs.

I'm more hopeful about Bayesian filters, because they evolve with the spam. So as spammers start using "c0ck" instead of "cock" to evade simple-minded spam filters based on individual words, Bayesian filters automatically notice. Indeed, "c0ck" is far more damning evidence than "cock", and Bayesian filters know precisely how much more.

Still, anyone who proposes a plan for spam filtering has to be able to answer the question: if the spammers knew exactly what you were doing, how well could they

get past you? For example, I think that if checksum-based spam filtering becomes a serious obstacle, the spammers will just switch to mad-lib techniques for generating message bodies.

To beat Bayesian filters, it would not be enough for spammers to make their emails unique or to stop using individual naughty words. They'd have to make their mails indistinguishable from your ordinary mail. And this I think would severely constrain them. Spam is mostly sales pitches, so unless your regular mail is all sales pitches, spams will inevitably have a different character. And the spammers would also, of course, have to change (and keep changing) their whole infrastructure, because otherwise the headers would look as bad to the Bayesian filters as ever, no matter what they did to the message body. I don't know enough about the infrastructure that spammers use to know how hard it would be to make the headers look innocent, but my guess is that it would be even harder than making the message look innocent.

Assuming they could solve the problem of the headers, the spam of the future will probably look something like this:

```
Hey there.  Thought you should check out the following:
http://www.27meg.com/foo
```

because that is about as much sales pitch as content-based filtering will leave the spammer room to make. (Indeed, it will be hard even to get this past filters, because if everything else in the email is neutral, the spam probability will hinge on the url, and it will take some effort to make that look neutral.)

Spammers range from businesses running so-called opt-in lists who don't even try to conceal their identities, to guys who hijack mail servers to send out spams promoting porn sites. If we use filtering to whittle their options down to mails like the one above, that should pretty much put the spammers on the "legitimate" end of the spectrum out of business; they feel obliged by various state laws to include boilerplate about why their spam is not spam, and how to cancel your "subscription," and that kind of text is easy to recognize.

(I used to think it was naive to believe that stricter laws would decrease spam. Now I think that while stricter laws may not decrease the amount of spam that spammers *send,* they can certainly help filters to decrease the amount of spam that recipients actually see.)

All along the spectrum, if you restrict the sales pitches spammers can make, you will inevitably tend to put them out of business. That word *business* is an important one to remember. The spammers are businessmen. They send spam because it works. It works because although the response rate is abominably low (at best 15 per million, vs 3000 per million for a catalog mailing), the cost, to them, is practically nothing. The cost is enormous for the recipients, about 5 man-weeks for each million recipients who spend a second to delete the spam, but the spammer doesn't have to pay that.

Sending spam does cost the spammer something, though. [2] So the lower we can get the response rate-- whether by filtering, or by using filters to force spammers to dilute their pitches-- the fewer businesses will find it worth their while to send spam.

The reason the spammers use the kinds of sales pitches that they do is to increase response rates. This is possibly even more disgusting than getting inside the mind of a spammer, but let's take a quick look inside the mind of someone who *responds* to a spam. This person is either astonishingly credulous or deeply in denial about their sexual interests. In either case, repulsive or idiotic as the spam seems to us, it is exciting to them. The spammers wouldn't say these things if they didn't sound exciting. And "thought you should check out the following" is just not going to have nearly the pull with the spam recipient as the kinds of things that spammers say now. Result: if it can't contain exciting sales pitches, spam becomes less effective as a marketing vehicle, and fewer businesses want to use it.

That is the big win in the end. I started writing spam filtering software because I didn't want have to look at the stuff anymore. But if we get good enough at filtering out spam, it will stop working, and the spammers will actually stop sending it.

— — —

Of all the approaches to fighting spam, from software to laws, I believe Bayesian filtering will be the single most effective. But I also think that the more different kinds of antispam efforts we undertake, the better, because any measure that constrains spammers will tend to make filtering easier. And even within the world of content-based filtering, I think it will be a good thing if there are many different kinds of software being used simultaneously. The more different filters there are, the harder it will be for spammers to tune spams to get through them.

Appendix: Examples of Filtering

Here is an example of a spam that arrived while I was writing this article. The fifteen most interesting words in this spam are:

```
qvp0045
indira
mx-05
intimail
$7500
freeyankeedom
cdo
bluefoxmedia
jpg
```

```
unsecured
platinum
3d0
qves
7c5
7c266675
```
The words are a mix of stuff from the headers and from the message body, which is typical of spam. Also typical of spam is that every one of these words has a spam probability, in my database, of .99. In fact there are more than fifteen words with probabilities of .99, and these are just the first fifteen seen.

Unfortunately that makes this email a boring example of the use of Bayes' Rule. To see an interesting variety of probabilities we have to look at this actually quite atypical spam.

The fifteen most interesting words in this spam, with their probabilities, are:

```
madam             0.99
promotion         0.99
republic          0.99
shortest          0.047225013
mandatory         0.047225013
standardization   0.07347802
sorry             0.08221981
supported         0.09019077
people's          0.09019077
enter             0.9075001
quality           0.8921298
organization      0.12454646
investment        0.8568143
very              0.14758544
valuable          0.82347786
```

This time the evidence is a mix of good and bad. A word like "shortest" is almost as much evidence for innocence as a word like "madam" or "promotion" is for guilt. But still the case for guilt is stronger. If you combine these numbers according to Bayes' Rule, the resulting probability is .9027.

"Madam" is obviously from spams beginning "Dear Sir or Madam." They're not very common, but the word "madam" *never* occurs in my legitimate email, and it's all about the ratio.

"Republic" scores high because it often shows up in Nigerian scam emails, and also occurs once or twice in spams referring to Korea and South Africa. You might say that it's an accident that it thus helps identify this spam. But I've found when examining spam probabilities that there are a lot of these accidents, and they have an uncanny tendency to push things in the right direction rather than the wrong one. In this case, it is not entirely a coincidence that the word "Republic" occurs in Nigerian scam emails and this spam. There is a whole class of dubious business propositions involving less developed countries, and these in turn are more likely to have names that specify explicitly (because they aren't) that they are republics. [3]

On the other hand, "enter" is a genuine miss. It occurs mostly in unsubscribe instructions, but here is used in a completely innocent way. Fortunately the statistical approach is fairly robust, and can tolerate quite a lot of misses before the results start to be thrown off.

For comparison, here is an example of that rare bird, a spam that gets through the filters. Why? Because by sheer chance it happens to be loaded with words that occur in my actual email:

```
perl        0.01
python      0.01
tcl         0.01
scripting   0.01
morris      0.01
graham      0.01491078
guarantee   0.9762507
cgi         0.9734398
paul        0.027040077
quite       0.030676773
pop3        0.042199217
various     0.06080265
prices      0.9359873
managed     0.06451222
difficult   0.071706355
```

There are a couple pieces of good news here. First, this mail probably wouldn't get through the filters of someone who didn't happen to specialize in programming languages and have a good friend called Morris. For the average user, all the top five words here would be neutral and would not contribute to the spam probability.

Second, I think filtering based on word pairs (see below) might well catch this one: "cost effective", "setup fee", "money back" -- pretty incriminating stuff. And of course if they continued to spam me (or a network I was part of), "Hostex" itself would be recognized as a spam term.

Finally, here is an innocent email. Its fifteen most interesting words are as follows:

```
continuation   0.01
describe       0.01
continuations  0.01
example        0.033600237
programming    0.05214485
i'm            0.055427782
examples       0.07972858
color          0.9189189
localhost      0.09883721
hi             0.116539136
california     0.84421706
same           0.15981844
spot           0.1654587
us-ascii       0.16804294
what           0.19212411
```

Most of the words here indicate the mail is an innocent one. There are two bad smelling words, "color" (spammers love colored fonts) and "California" (which occurs in testimonials and also in menus in forms), but they are not enough to outweigh obviously innocent words like "continuation" and "example".

It's interesting that "describe" rates as so thoroughly innocent. It hasn't occurred in a single one of my 4000 spams. The data turns out to be full of such surprises. One of the things you learn when you analyze spam texts is how narrow a subset of the language spammers operate in. It's that fact, together with the equally characteristic vocabulary of any individual user's mail, that makes Bayesian filtering a good bet.

Appendix: More Ideas

One idea that I haven't tried yet is to filter based on word pairs, or even triples, rather than individual words. This should yield a much sharper estimate of the probability. For example, in my current database, the word "offers" has a probability of .96. If you based the probabilities on word pairs, you'd end up with "special offers" and "valuable offers" having probabilities of .99 and, say, "approach offers" (as in "this approach offers") having a probability of .1 or less.

The reason I haven't done this is that filtering based on individual words already works so well. But it does mean that there is room to tighten the filters if spam gets harder to detect. (Curiously, a filter based on word pairs would be in effect a Markov-chaining text generator running in reverse.)

Specific spam features (e.g. not seeing the recipient's address in the to: field) do of course have value in recognizing spam. They can be considered in this algorithm by treating them as virtual words. I'll probably do this in future versions, at least for a handful of the most egregious spam indicators. Feature-recognizing spam filters are right in many details; what they lack is an overall discipline for combining evidence.

Recognizing nonspam features may be more important than recognizing spam features. False positives are such a worry that they demand extraordinary measures. I will probably in future versions add a second level of testing designed specifically to avoid false positives. If a mail triggers this second level of filters it will be accepted even if its spam probability is above the threshold.

I don't expect this second level of filtering to be Bayesian. It will inevitably be not only ad hoc, but based on guesses, because the number of false positives will not tend to be large enough to notice patterns. (It is just as well, anyway, if a backup system doesn't rely on the same technology as the primary system.)

Another thing I may try in the future is to focus extra attention on specific parts of the email. For example, about 95% of current spam includes the url of a site they want you to visit. (The remaining 5% want you to call a phone number, reply by email or to a US mail address, or in a few cases to buy a certain stock.) The url is

in such cases practically enough by itself to determine whether the email is spam.

Domain names differ from the rest of the text in a (non-German) email in that they often consist of several words stuck together. Though computationally expensive in the general case, it might be worth trying to decompose them. If a filter has never seen the token "xxxporn" before it will have an individual spam probability of .4, whereas "xxx" and "porn" individually have probabilities (in my corpus) of .9889 and .99 respectively, and a combined probability of .9998.

I expect decomposing domain names to become more important as spammers are gradually forced to stop using incriminating words in the text of their messages. (A url with an ip address is of course an extremely incriminating sign, except in the mail of a few sysadmins.)

It might be a good idea to have a cooperatively maintained list of urls promoted by spammers. We'd need a trust metric of the type studied by Raph Levien to prevent malicious or incompetent submissions, but if we had such a thing it would provide a boost to any filtering software. It would also be a convenient basis for boycotts.

Another way to test dubious urls would be to send out a crawler to look at the site before the user looked at the email mentioning it. You could use a Bayesian filter to rate the site just as you would an email, and whatever was found on the site could be included in calculating the probability of the email being a spam. A url that led to a redirect would of course be especially suspicious.

One cooperative project that I think really would be a good idea would be to accumulate a giant corpus of spam. A large, clean corpus is the key to making Bayesian filtering work well. Bayesian filters could actually use the corpus as input. But such a corpus would be useful for other kinds of filters too, because it could be used to test them.

Creating such a corpus poses some technical problems. We'd need trust metrics to prevent malicious or incompetent submissions, of course. We'd also need ways of erasing personal information (not just to-addresses and ccs, but also e.g. the arguments to unsubscribe urls, which often encode the to-address) from mails in the corpus. If anyone wants to take on this project, it would be a good thing for the world.

Appendix: Defining Spam

I think there is a rough consensus on what spam is, but it would be useful to have an explicit definition. We'll need to do this if we want to establish a central corpus of spam, or even to compare spam filtering rates meaningfully.

To start with, spam is not unsolicited commercial email. If someone in my neighborhood heard that I was looking for an old Raleigh three-speed in good condition, and sent me an email offering to sell me one, I'd be delighted, and yet this email would be both commercial and unsolicited. The defining feature of spam

(in fact, its *raison d'etre*) is not that it is unsolicited, but that it is automated.

It is merely incidental, too, that spam is usually commercial. If someone started sending mass email to support some political cause, for example, it would be just as much spam as email promoting a porn site.

I propose we define spam as **unsolicited automated email**. This definition thus includes some email that many legal definitions of spam don't. Legal definitions of spam, influenced presumably by lobbyists, tend to exclude mail sent by companies that have an "existing relationship" with the recipient. But buying something from a company, for example, does not imply that you have solicited ongoing email from them. If I order something from an online store, and they then send me a stream of spam, it's still spam.

Companies sending spam often give you a way to "unsubscribe," or ask you to go to their site and change your "account preferences" if you want to stop getting spam. This is not enough to stop the mail from being spam. Not opting out is not the same as opting in. Unless the recipient explicitly checked a clearly labelled box (whose default was no) asking to receive the email, then it is spam.

In some business relationships, you do implicitly solicit certain kinds of mail. When you order online, I think you implicitly solicit a receipt, and notification when the order ships. I don't mind when Verisign sends me mail warning that a domain name is about to expire (at least, if they are the [actual registrar](#) for it). But when Verisign sends me email offering a FREE Guide to Building My E-Commerce Web Site, that's spam.

Notes:

[1] The examples in this article are translated into Common Lisp for, believe it or not, greater accessibility. The application described here is one that we wrote in order to test a new Lisp dialect called [Arc](#) that is not yet released.

[2] Currently the lowest rate seems to be about $200 to send a million spams. That's very cheap, 1/50th of a cent per spam. But filtering out 95% of spam, for example, would increase the spammers' cost to reach a given audience by a factor of 20. Few can have margins big enough to absorb that.

[3] As a rule of thumb, the more qualifiers there are before the name of a country, the more corrupt the rulers. A country called The Socialist People's Democratic Republic of X is probably the last place in the world you'd want to live.

Thanks to Sarah Harlin for reading drafts of this; Daniel Giffin (who is also writing the production Arc interpreter) for several good ideas about filtering and for creating our mail infrastructure; Robert Morris, Trevor Blackwell and Erann Gat for many discussions about spam; Raph Levien for advice about trust metrics; and Chip Coldwell and Sam Steingold for advice about statistics.

You'll find this essay and 14 others in ***Hackers & Painters***.

More Info:

- [Plan for Spam FAQ](#)

- [Better Bayesian Filtering](#)

- [Filters that Fight Back](#)

- [Will Filters Kill Spam?](#)

- [Japanese Translation](#)

- [Spanish Translation](#)

- [Chinese Translation](#)

- [Probability](#)

- [Spam is Different](#)

- [Filters vs. Blacklists](#)

- [Trust Metrics](#)

- [Filtering Research](#)

- [Microsoft Patent](#)

- [Slashdot Article](#)

- [The Wrong Way](#)

- [LWN: Filter Comparison](#)

- [CRM114 gets 99.87%](#)

Design and Research

January 2003

(This article is derived from a keynote talk at the fall 2002 meeting of NEPLS.)

Visitors to this country are often surprised to find that Americans like to begin a conversation by asking "what do you do?" I've never liked this question. I've rarely had a neat answer to it. But I think I have finally solved the problem. Now, when someone asks me what I do, I look them straight in the eye and say "I'm designing a [new dialect of Lisp](#)." I recommend this answer to anyone who doesn't like being asked what they do. The conversation will turn immediately to other topics.

I don't consider myself to be doing research on programming languages. I'm just designing one, in the same way that someone might design a building or a chair or a new typeface. I'm not trying to discover anything new. I just want to make a language that will be good to program in. In some ways, this assumption makes life a lot easier.

The difference between design and research seems to be a question of new versus good. Design doesn't have to be new, but it has to be good. Research doesn't have to be good, but it has to be new. I think these two paths converge at the top: the best design surpasses its predecessors by using new ideas, and the best research solves problems that are not only new, but actually worth solving. So ultimately we're aiming for the same destination, just approaching it from different directions.

What I'm going to talk about today is what your target looks like from the back. What do you do differently when you treat programming languages as a design problem instead of a research topic?

The biggest difference is that you focus more on the user. Design begins by asking, who is this for and what do they need from it? A good architect, for example, does not begin by creating a design that he then imposes on the users, but by studying the intended users and figuring out what they need.

Notice I said "what they need," not "what they want." I don't mean to give the impression that working as a designer means working as a sort of short-order cook, making whatever the client tells you to. This varies from field to field in the

arts, but I don't think there is any field in which the best work is done by the people who just make exactly what the customers tell them to.

The customer *is* always right in the sense that the measure of good design is how well it works for the user. If you make a novel that bores everyone, or a chair that's horribly uncomfortable to sit in, then you've done a bad job, period. It's no defense to say that the novel or the chair is designed according to the most advanced theoretical principles.

And yet, making what works for the user doesn't mean simply making what the user tells you to. Users don't know what all the choices are, and are often mistaken about what they really want.

The answer to the paradox, I think, is that you have to design for the user, but you have to design what the user needs, not simply what he says he wants. It's much like being a doctor. You can't just treat a patient's symptoms. When a patient tells you his symptoms, you have to figure out what's actually wrong with him, and treat that.

This focus on the user is a kind of axiom from which most of the practice of good design can be derived, and around which most design issues center.

If good design must do what the user needs, who is the user? When I say that design must be for users, I don't mean to imply that good design aims at some kind of lowest common denominator. You can pick any group of users you want. If you're designing a tool, for example, you can design it for anyone from beginners to experts, and what's good design for one group might be bad for another. The point is, you have to pick some group of users. I don't think you can even talk about good or bad design except with reference to some intended user.

You're most likely to get good design if the intended users include the designer himself. When you design something for a group that doesn't include you, it tends to be for people you consider to be less sophisticated than you, not more sophisticated.

That's a problem, because looking down on the user, however benevolently, seems inevitably to corrupt the designer. I suspect that very few housing projects in the US were designed by architects who expected to live in them. You can see the same thing in programming languages. C, Lisp, and Smalltalk were created for their own designers to use. Cobol, Ada, and Java, were created for other people to use.

If you think you're designing something for idiots, the odds are that you're not designing something good, even for idiots.

Even if you're designing something for the most sophisticated users, though, you're still designing for humans. It's different in research. In math you don't choose abstractions because they're easy for humans to understand; you choose whichever make the proof shorter. I think this is true for the sciences generally. Scientific ideas are not meant to be ergonomic.

Over in the arts, things are very different. Design is all about people. The human body is a strange thing, but when you're designing a chair, that's what you're designing for, and there's no way around it. All the arts have to pander to the interests and limitations of humans. In painting, for example, all other things being equal a painting with people in it will be more interesting than one without. It is not merely an accident of history that the great paintings of the Renaissance are all full of people. If they hadn't been, painting as a medium wouldn't have the prestige that it does.

Like it or not, programming languages are also for people, and I suspect the human brain is just as lumpy and idiosyncratic as the human body. Some ideas are easy for people to grasp and some aren't. For example, we seem to have a very limited capacity for dealing with detail. It's this fact that makes programing languages a good idea in the first place; if we could handle the detail, we could just program in machine language.

Remember, too, that languages are not primarily a form for finished programs, but something that programs have to be developed in. Anyone in the arts could tell you that you might want different mediums for the two situations. Marble, for example, is a nice, durable medium for finished ideas, but a hopelessly inflexible one for developing new ideas.

A program, like a proof, is a pruned version of a tree that in the past has had false starts branching off all over it. So the test of a language is not simply how clean the finished program looks in it, but how clean the path to the finished program was. A design choice that gives you elegant finished programs may not give you an elegant design process. For example, I've written a few macro-defining macros full of nested backquotes that look now like little gems, but writing them took hours of the ugliest trial and error, and frankly, I'm still not entirely sure they're correct.

We often act as if the test of a language were how good finished programs look in it. It seems so convincing when you see the same program written in two languages, and one version is much shorter. When you approach the problem from the direction of the arts, you're less likely to depend on this sort of test. You don't want to end up with a programming language like marble.

For example, it is a huge win in developing software to have an interactive toplevel, what in Lisp is called a read-eval-print loop. And when you have one this has real effects on the design of the language. It would not work well for a language where you have to declare variables before using them, for example. When you're just typing expressions into the toplevel, you want to be able to set x

to some value and then start doing things to x. You don't want to have to declare the type of x first. You may dispute either of the premises, but if a language has to have a toplevel to be convenient, and mandatory type declarations are incompatible with a toplevel, then no language that makes type declarations mandatory could be convenient to program in.

In practice, to get good design you have to get close, and stay close, to your users. You have to calibrate your ideas on actual users constantly, especially in the beginning. One of the reasons Jane Austen's novels are so good is that she read them out loud to her family. That's why she never sinks into self-indulgently arty descriptions of landscapes, or pretentious philosophizing. (The philosophy's there, but it's woven into the story instead of being pasted onto it like a label.) If you open an average "literary" novel and imagine reading it out loud to your friends as something you'd written, you'll feel all too keenly what an imposition that kind of thing is upon the reader.

In the software world, this idea is known as Worse is Better. Actually, there are several ideas mixed together in the concept of Worse is Better, which is why people are still arguing about whether worse is actually better or not. But one of the main ideas in that mix is that if you're building something new, you should get a prototype in front of users as soon as possible.

The alternative approach might be called the Hail Mary strategy. Instead of getting a prototype out quickly and gradually refining it, you try to create the complete, finished, product in one long touchdown pass. As far as I know, this is a recipe for disaster. Countless startups destroyed themselves this way during the Internet bubble. I've never heard of a case where it worked.

What people outside the software world may not realize is that Worse is Better is found throughout the arts. In drawing, for example, the idea was discovered during the Renaissance. Now almost every drawing teacher will tell you that the right way to get an accurate drawing is not to work your way slowly around the contour of an object, because errors will accumulate and you'll find at the end that the lines don't meet. Instead you should draw a few quick lines in roughly the right place, and then gradually refine this initial sketch.

In most fields, prototypes have traditionally been made out of different materials. Typefaces to be cut in metal were initially designed with a brush on paper. Statues to be cast in bronze were modelled in wax. Patterns to be embroidered on tapestries were drawn on paper with ink wash. Buildings to be constructed from stone were tested on a smaller scale in wood.

What made oil paint so exciting, when it first became popular in the fifteenth century, was that you could actually make the finished work *from* the prototype. You could make a preliminary drawing if you wanted to, but you weren't held to it; you could work out all the details, and even make major changes, as you finished

the painting.

You can do this in software too. A prototype doesn't have to be just a model; you can refine it into the finished product. I think you should always do this when you can. It lets you take advantage of new insights you have along the way. But perhaps even more important, it's good for morale.

Morale is key in design. I'm surprised people don't talk more about it. One of my first drawing teachers told me: if you're bored when you're drawing something, the drawing will look boring. For example, suppose you have to draw a building, and you decide to draw each brick individually. You can do this if you want, but if you get bored halfway through and start making the bricks mechanically instead of observing each one, the drawing will look worse than if you had merely suggested the bricks.

Building something by gradually refining a prototype is good for morale because it keeps you engaged. In software, my rule is: always have working code. If you're writing something that you'll be able to test in an hour, then you have the prospect of an immediate reward to motivate you. The same is true in the arts, and particularly in oil painting. Most painters start with a blurry sketch and gradually refine it. If you work this way, then in principle you never have to end the day with something that actually looks unfinished. Indeed, there is even a saying among painters: "A painting is never finished, you just stop working on it." This idea will be familiar to anyone who has worked on software.

Morale is another reason that it's hard to design something for an unsophisticated user. It's hard to stay interested in something you don't like yourself. To make something good, you have to be thinking, "wow, this is really great," not "what a piece of shit; those fools will love it."

Design means making things for humans. But it's not just the user who's human. The designer is human too.

Notice all this time I've been talking about "the designer." Design usually has to be under the control of a single person to be any good. And yet it seems to be possible for several people to collaborate on a research project. This seems to me one of the most interesting differences between research and design.

There have been famous instances of collaboration in the arts, but most of them seem to have been cases of molecular bonding rather than nuclear fusion. In an opera it's common for one person to write the libretto and another to write the music. And during the Renaissance, journeymen from northern Europe were often employed to do the landscapes in the backgrounds of Italian paintings. But these aren't true collaborations. They're more like examples of Robert Frost's "good

fences make good neighbors." You can stick instances of good design together, but within each individual project, one person has to be in control.

I'm not saying that good design requires that one person think of everything. There's nothing more valuable than the advice of someone whose judgement you trust. But after the talking is done, the decision about what to do has to rest with one person.

Why is it that research can be done by collaborators and design can't? This is an interesting question. I don't know the answer. Perhaps, if design and research converge, the best research is also good design, and in fact can't be done by collaborators. A lot of the most famous scientists seem to have worked alone. But I don't know enough to say whether there is a pattern here. It could be simply that many famous scientists worked when collaboration was less common.

Whatever the story is in the sciences, true collaboration seems to be vanishingly rare in the arts. Design by committee is a synonym for bad design. Why is that so? Is there some way to beat this limitation?

I'm inclined to think there isn't-- that good design requires a dictator. One reason is that good design has to be all of a piece. Design is not just for humans, but for individual humans. If a design represents an idea that fits in one person's head, then the idea will fit in the user's head too.

Related:

■

[Japanese Translation](#)

■

[Taste for Makers](#)

■

[Romanian Translation](#)

[Spanish Translation](#)

Better Bayesian Filtering

January 2003

(This article was given as a talk at the 2003 Spam Conference. It describes the work I've done to improve the performance of the algorithm described in A Plan for Spam, and what I plan to do in the future.)

The first discovery I'd like to present here is an algorithm for lazy evaluation of research papers. Just write whatever you want and don't cite any previous work, and indignant readers will send you references to all the papers you should have cited. I discovered this algorithm after ``A Plan for Spam'' [1] was on Slashdot.

Spam filtering is a subset of text classification, which is a well established field, but the first papers about Bayesian spam filtering per se seem to have been two given at the same conference in 1998, one by Pantel and Lin [2], and another by a group from Microsoft Research [3].

When I heard about this work I was a bit surprised. If people had been onto Bayesian filtering four years ago, why wasn't everyone using it? When I read the papers I found out why. Pantel and Lin's filter was the more effective of the two, but it only caught 92% of spam, with 1.16% false positives.

When I tried writing a Bayesian spam filter, it caught 99.5% of spam with less than .03% false positives [4]. It's always alarming when two people trying the same experiment get widely divergent results. It's especially alarming here because those two sets of numbers might yield opposite conclusions. Different users have different requirements, but I think for many people a filtering rate of 92% with 1.16% false positives means that filtering is not an acceptable solution, whereas 99.5% with less than .03% false positives means that it is.

So why did we get such different numbers? I haven't tried to reproduce Pantel and Lin's results, but from reading the paper I see five things that probably account for the difference.

One is simply that they trained their filter on very little data: 160 spam and 466 nonspam mails. Filter performance should still be climbing with data sets that small. So their numbers may not even be an accurate measure of the performance of their algorithm, let alone of Bayesian spam filtering in general.

But I think the most important difference is probably that they ignored message headers. To anyone who has worked on spam filters, this will seem a perverse decision. And yet in the very first filters I tried writing, I ignored the headers too. Why? Because I wanted to keep the problem neat. I didn't know much about mail headers then, and they seemed to me full of random stuff. There is a lesson here for filter writers: don't ignore data. You'd think this lesson would be too obvious to mention, but I've had to learn it several times.

Third, Pantel and Lin stemmed the tokens, meaning they reduced e.g. both ``mailing'' and ``mailed'' to the root ``mail''. They may have felt they were forced to do this by the small size of their corpus, but if so this is a kind of premature optimization.

Fourth, they calculated probabilities differently. They used all the tokens, whereas I only use the 15 most significant. If you use all the tokens you'll tend to miss longer spams, the type where someone tells you their life story up to the point where they got rich from some multilevel marketing scheme. And such an algorithm would be easy for spammers to spoof: just add a big chunk of random text to counterbalance the spam terms.

Finally, they didn't bias against false positives. I think any spam filtering algorithm ought to have a convenient knob you can twist to decrease the false positive rate at the expense of the filtering rate. I do this by counting the occurrences of tokens in the nonspam corpus double.

I don't think it's a good idea to treat spam filtering as a straight text classification problem. You can use text classification techniques, but solutions can and should reflect the fact that the text is email, and spam in particular. Email is not just text; it has structure. Spam filtering is not just classification, because false positives are so much worse than false negatives that you should treat them as a different kind of error. And the source of error is not just random variation, but a live human spammer working actively to defeat your filter.

Tokens

Another project I heard about after the Slashdot article was Bill Yerazunis' CRM114 [5]. This is the counterexample to the design principle I just mentioned. It's a straight text classifier, but such a stunningly effective one that it manages to filter spam almost perfectly without even knowing that's what it's doing.

Once I understood how CRM114 worked, it seemed inevitable that I would eventually have to move from filtering based on single words to an approach like this. But first, I thought, I'll see how far I can get with single words. And the answer is, surprisingly far.

Mostly I've been working on smarter tokenization. On current spam, I've been able to achieve filtering rates that approach CRM114's. These techniques are mostly orthogonal to Bill's; an optimal solution might incorporate both.

``A Plan for Spam'' uses a very simple definition of a token. Letters, digits, dashes, apostrophes, and dollar signs are constituent characters, and everything else is a token separator. I also ignored case.

Now I have a more complicated definition of a token:

1. Case is preserved.

2. Exclamation points are constituent characters.

3. Periods and commas are constituents if they occur between two digits. This lets me get ip addresses and prices intact.

4. A price range like $20-25 yields two tokens, $20 and $25.

5. Tokens that occur within the To, From, Subject, and Return-Path lines, or within urls, get marked accordingly. E.g. ``foo'' in the Subject line becomes ``Subject*foo''. (The asterisk could be any character you don't allow as a constituent.)

Such measures increase the filter's vocabulary, which makes it more discriminating. For example, in the current filter, ``free'' in the Subject line has a spam probability of 98%, whereas the same token in the body has a spam probability of only 65%.

Here are some of the current probabilities [6]:

```
Subject*FREE        0.9999
free!!              0.9999
To*free             0.9998
Subject*free        0.9782
free!               0.9199
Free                0.9198
Url*free            0.9091
FREE                0.8747
From*free           0.7636
free                0.6546
```

In the Plan for Spam filter, all these tokens would have had the same probability, .7602. That filter recognized about 23,000 tokens. The current one recognizes about 187,000.

The disadvantage of having a larger universe of tokens is that there is more chance of misses. Spreading your corpus out over more tokens has the same effect as making it smaller. If you consider exclamation points as constituents, for example, then you could end up not having a spam probability for free with seven exclamation points, even though you know that free with just two exclamation points has a probability of 99.99%.

One solution to this is what I call degeneration. If you can't find an exact match for

a token, treat it as if it were a less specific version. I consider terminal exclamation points, uppercase letters, and occurring in one of the five marked contexts as making a token more specific. For example, if I don't find a probability for ``Subject*free!'', I look for probabilities for ``Subject*free'', ``free!'', and ``free'', and take whichever one is farthest from .5.

Here are the alternatives [7] considered if the filter sees ``FREE!!!'' in the Subject line and doesn't have a probability for it.

```
Subject*Free!!!
Subject*free!!!
Subject*FREE!
Subject*Free!
Subject*free!
Subject*FREE
Subject*Free
Subject*free
FREE!!!
Free!!!
free!!!
FREE!
Free!
free!
FREE
Free
free
```

If you do this, be sure to consider versions with initial caps as well as all uppercase and all lowercase. Spams tend to have more sentences in imperative mood, and in those the first word is a verb. So verbs with initial caps have higher spam probabilities than they would in all lowercase. In my filter, the spam probability of ``Act'' is 98% and for ``act'' only 62%.

If you increase your filter's vocabulary, you can end up counting the same word multiple times, according to your old definition of ``same''. Logically, they're not the same token anymore. But if this still bothers you, let me add from experience that the words you seem to be counting multiple times tend to be exactly the ones you'd want to.

Another effect of a larger vocabulary is that when you look at an incoming mail you find more interesting tokens, meaning those with probabilities far from .5. I use the 15 most interesting to decide if mail is spam. But you can run into a problem when you use a fixed number like this. If you find a lot of maximally interesting tokens, the result can end up being decided by whatever random factor determines the ordering of equally interesting tokens. One way to deal with this is to treat some as more interesting than others.

For example, the token ``dalco'' occurs 3 times in my spam corpus and never in my legitimate corpus. The token ``Url*optmails'' (meaning ``optmails'' within a url) occurs 1223 times. And yet, as I used to calculate probabilities for tokens, both would have the same spam probability, the threshold of .99.

That doesn't feel right. There are theoretical arguments for giving these two tokens substantially different probabilities (Pantel and Lin do), but I haven't tried that yet. It does seem at least that if we find more than 15 tokens that only occur in one corpus or the other, we ought to give priority to the ones that occur a lot. So now there are two threshold values. For tokens that occur only in the spam corpus, the probability is .9999 if they occur more than 10 times and .9998 otherwise. Ditto at the other end of the scale for tokens found only in the legitimate corpus.

I may later scale token probabilities substantially, but this tiny amount of scaling at least ensures that tokens get sorted the right way.

Another possibility would be to consider not just 15 tokens, but all the tokens over a certain threshold of interestingness. Steven Hauser does this in his statistical spam filter [8]. If you use a threshold, make it very high, or spammers could spoof you by packing messages with more innocent words.

Finally, what should one do about html? I've tried the whole spectrum of options, from ignoring it to parsing it all. Ignoring html is a bad idea, because it's full of useful spam signs. But if you parse it all, your filter might degenerate into a mere html recognizer. The most effective approach seems to be the middle course, to notice some tokens but not others. I look at a, img, and font tags, and ignore the rest. Links and images you should certainly look at, because they contain urls.

I could probably be smarter about dealing with html, but I don't think it's worth putting a lot of time into this. Spams full of html are easy to filter. The smarter spammers already avoid it. So performance in the future should not depend much on how you deal with html.

Performance

Between December 10 2002 and January 10 2003 I got about 1750 spams. Of these, 4 got through. That's a filtering rate of about 99.75%.

Two of the four spams I missed got through because they happened to use words that occur often in my legitimate email.

The third was one of those that exploit an insecure cgi script to send mail to third parties. They're hard to filter based just on the content because the headers are innocent and they're careful about the words they use. Even so I can usually catch them. This one squeaked by with a probability of .88, just under the threshold of .9.

Of course, looking at multiple token sequences would catch it easily. ``Below is the result of your feedback form'' is an instant giveaway.

The fourth spam was what I call a spam-of-the-future, because this is what I expect spam to evolve into: some completely neutral text followed by a url. In this

case it was was from someone saying they had finally finished their homepage and would I go look at it. (The page was of course an ad for a porn site.)

If the spammers are careful about the headers and use a fresh url, there is nothing in spam-of-the-future for filters to notice. We can of course counter by sending a crawler to look at the page. But that might not be necessary. The response rate for spam-of-the-future must be low, or everyone would be doing it. If it's low enough, it won't pay for spammers to send it, and we won't have to work too hard on filtering it.

Now for the really shocking news: during that same one-month period I got *three* false positives.

In a way it's a relief to get some false positives. When I wrote ``A Plan for Spam'' I hadn't had any, and I didn't know what they'd be like. Now that I've had a few, I'm relieved to find they're not as bad as I feared. False positives yielded by statistical filters turn out to be mails that sound a lot like spam, and these tend to be the ones you would least mind missing [9].

Two of the false positives were newsletters from companies I've bought things from. I never asked to receive them, so arguably they were spams, but I count them as false positives because I hadn't been deleting them as spams before. The reason the filters caught them was that both companies in January switched to commercial email senders instead of sending the mails from their own servers, and both the headers and the bodies became much spammier.

The third false positive was a bad one, though. It was from someone in Egypt and written in all uppercase. This was a direct result of making tokens case sensitive; the Plan for Spam filter wouldn't have caught it.

It's hard to say what the overall false positive rate is, because we're up in the noise, statistically. Anyone who has worked on filters (at least, effective filters) will be aware of this problem. With some emails it's hard to say whether they're spam or not, and these are the ones you end up looking at when you get filters really tight. For example, so far the filter has caught two emails that were sent to my address because of a typo, and one sent to me in the belief that I was someone else. Arguably, these are neither my spam nor my nonspam mail.

Another false positive was from a vice president at Virtumundo. I wrote to them pretending to be a customer, and since the reply came back through Virtumundo's mail servers it had the most incriminating headers imaginable. Arguably this isn't a real false positive either, but a sort of Heisenberg uncertainty effect: I only got it because I was writing about spam filtering.

Not counting these, I've had a total of five false positives so far, out of about 7740 legitimate emails, a rate of .06%. The other two were a notice that something I bought was back-ordered, and a party reminder from Evite.

I don't think this number can be trusted, partly because the sample is so small, and partly because I think I can fix the filter not to catch some of these.

False positives seem to me a different kind of error from false negatives. Filtering rate is a measure of performance. False positives I consider more like bugs. I approach improving the filtering rate as optimization, and decreasing false positives as debugging.

So these five false positives are my bug list. For example, the mail from Egypt got nailed because the uppercase text made it look to the filter like a Nigerian spam. This really is kind of a bug. As with html, the email being all uppercase is really conceptually *one* feature, not one for each word. I need to handle case in a more sophisticated way.

So what to make of this .06%? Not much, I think. You could treat it as an upper bound, bearing in mind the small sample size. But at this stage it is more a measure of the bugs in my implementation than some intrinsic false positive rate of Bayesian filtering.

Future

What next? Filtering is an optimization problem, and the key to optimization is profiling. Don't try to guess where your code is slow, because you'll guess wrong. *Look* at where your code is slow, and fix that. In filtering, this translates to: look at the spams you miss, and figure out what you could have done to catch them.

For example, spammers are now working aggressively to evade filters, and one of the things they're doing is breaking up and misspelling words to prevent filters from recognizing them. But working on this is not my first priority, because I still have no trouble catching these spams [10].

There are two kinds of spams I currently do have trouble with. One is the type that pretends to be an email from a woman inviting you to go chat with her or see her profile on a dating site. These get through because they're the one type of sales pitch you can make without using sales talk. They use the same vocabulary as ordinary email.

The other kind of spams I have trouble filtering are those from companies in e.g. Bulgaria offering contract programming services. These get through because I'm a programmer too, and the spams are full of the same words as my real mail.

I'll probably focus on the personal ad type first. I think if I look closer I'll be able to find statistical differences between these and my real mail. The style of writing is certainly different, though it may take multiword filtering to catch that. Also, I notice they tend to repeat the url, and someone including a url in a legitimate mail wouldn't do that [11].

The outsourcing type are going to be hard to catch. Even if you sent a crawler to

the site, you wouldn't find a smoking statistical gun. Maybe the only answer is a central list of domains advertised in spams [12]. But there can't be that many of this type of mail. If the only spams left were unsolicited offers of contract programming services from Bulgaria, we could all probably move on to working on something else.

Will statistical filtering actually get us to that point? I don't know. Right now, for me personally, spam is not a problem. But spammers haven't yet made a serious effort to spoof statistical filters. What will happen when they do?

I'm not optimistic about filters that work at the network level [13]. When there is a static obstacle worth getting past, spammers are pretty efficient at getting past it. There is already a company called Assurance Systems that will run your mail through Spamassassin and tell you whether it will get filtered out.

Network-level filters won't be completely useless. They may be enough to kill all the "opt-in" spam, meaning spam from companies like Virtumundo and Equalamail who claim that they're really running opt-in lists. You can filter those based just on the headers, no matter what they say in the body. But anyone willing to falsify headers or use open relays, presumably including most porn spammers, should be able to get some message past network-level filters if they want to. (By no means the message they'd like to send though, which is something.)

The kind of filters I'm optimistic about are ones that calculate probabilities based on each individual user's mail. These can be much more effective, not only in avoiding false positives, but in filtering too: for example, finding the recipient's email address base-64 encoded anywhere in a message is a very good spam indicator.

But the real advantage of individual filters is that they'll all be different. If everyone's filters have different probabilities, it will make the spammers' optimization loop, what programmers would call their edit-compile-test cycle, appallingly slow. Instead of just tweaking a spam till it gets through a copy of some filter they have on their desktop, they'll have to do a test mailing for each tweak. It would be like programming in a language without an interactive toplevel, and I wouldn't wish that on anyone.

Notes

[1] Paul Graham. ``A Plan for Spam.'' August 2002.
http://paulgraham.com/spam.html.

Probabilities in this algorithm are calculated using a degenerate case of Bayes' Rule. There are two simplifying assumptions: that the probabilities of features (i.e. words) are independent, and that we know nothing about the prior probability of an email being spam.

The first assumption is widespread in text classification. Algorithms that use it are called ``naive Bayesian.''

The second assumption I made because the proportion of spam in my incoming mail fluctuated so much from day to day (indeed, from hour to hour) that the overall prior ratio seemed worthless as a predictor. If you assume that P(spam) and P(nonspam) are both .5, they cancel out and you can remove them from the formula.

If you were doing Bayesian filtering in a situation where the ratio of spam to nonspam was consistently very high or (especially) very low, you could probably improve filter performance by incorporating prior probabilities. To do this right you'd have to track ratios by time of day, because spam and legitimate mail volume both have distinct daily patterns.

[2] Patrick Pantel and Dekang Lin. ``SpamCop-- A Spam Classification & Organization Program.'' Proceedings of AAAI-98 Workshop on Learning for Text Categorization.

[3] Mehran Sahami, Susan Dumais, David Heckerman and Eric Horvitz. ``A Bayesian Approach to Filtering Junk E-Mail.'' Proceedings of AAAI-98 Workshop on Learning for Text Categorization.

[4] At the time I had zero false positives out of about 4,000 legitimate emails. If the next legitimate email was a false positive, this would give us .03%. These false positive rates are untrustworthy, as I explain later. I quote a number here only to emphasize that whatever the false positive rate is, it is less than 1.16%.

[5] Bill Yerazunis. ``Sparse Binary Polynomial Hash Message Filtering and The CRM114 Discriminator.'' Proceedings of 2003 Spam Conference.

[6] In ``A Plan for Spam'' I used thresholds of .99 and .01. It seems justifiable to use thresholds proportionate to the size of the corpora. Since I now have on the order of 10,000 of each type of mail, I use .9999 and .0001.

[7] There is a flaw here I should probably fix. Currently, when ``Subject*foo'' degenerates to just ``foo'', what that means is you're getting the stats for occurrences of ``foo'' in the body or header lines other than those I mark. What I should do is keep track of statistics for ``foo'' overall as well as specific versions, and degenerate from ``Subject*foo'' not to ``foo'' but to ``Anywhere*foo''. Ditto for case: I should degenerate from uppercase to any-case, not lowercase.

It would probably be a win to do this with prices too, e.g. to degenerate from ``$129.99'' to ``$--9.99'', ``$--.99'', and ``$--''.

You could also degenerate from words to their stems, but this would probably only improve filtering rates early on when you had small corpora.

[8] Steven Hauser. ``Statistical Spam Filter Works for Me.''
http://www.sofbot.com.

[9] False positives are not all equal, and we should remember this when comparing techniques for stopping spam. Whereas many of the false positives caused by filters will be near-spams that you wouldn't mind missing, false positives caused by blacklists, for example, will be just mail from people who chose the wrong ISP. In both cases you catch mail that's near spam, but for blacklists nearness is physical, and for filters it's textual.

[10] If spammers get good enough at obscuring tokens for this to be a problem, we can respond by simply removing whitespace, periods, commas, etc. and using a dictionary to pick the words out of the resulting sequence. And of course finding words this way that weren't visible in the original text would in itself be evidence of spam.

Picking out the words won't be trivial. It will require more than just reconstructing word boundaries; spammers both add (``xHot nPorn cSite'') and omit (``P#rn'') letters. Vision research may be useful here, since human vision is the limit that such tricks will approach.

[11] In general, spams are more repetitive than regular email. They want to pound that message home. I currently don't allow duplicates in the top 15 tokens, because you could get a false positive if the sender happens to use some bad word multiple times. (In my current filter, ``dick'' has a spam probabilty of .9999, but it's also a name.) It seems we should at least notice duplication though, so I may try allowing up to two of each token, as Brian Burton does in SpamProbe.

[12] This is what approaches like Brightmail's will degenerate into once spammers are pushed into using mad-lib techniques to generate everything else in the message.

[13] It's sometimes argued that we should be working on filtering at the network level, because it is more efficient. What people usually mean when they say this is: we currently filter at the network level, and we don't want to start over from scratch. But you can't dictate the problem to fit your solution.

Historically, scarce-resource arguments have been the losing side in debates about software design. People only tend to use them to justify choices (inaction in particular) made for other reasons.

Related:

- [A Plan for Spam](#)

- [Plan for Spam FAQ](#)

- [2003 Spam Conference Proceedings](#)

- [Japanese Translation](#)

- [Chinese Translation](#)

- [Test of These Suggestions](#)

Why Nerds are Unpopular

February 2003

When we were in junior high school, my friend Rich and I made a map of the school lunch tables according to popularity. This was easy to do, because kids only ate lunch with others of about the same popularity. We graded them from A to E. A tables were full of football players and cheerleaders and so on. E tables contained the kids with mild cases of Down's Syndrome, what in the language of the time we called "retards."

We sat at a D table, as low as you could get without looking physically different. We were not being especially candid to grade ourselves as D. It would have taken a deliberate lie to say otherwise. Everyone in the school knew exactly how popular everyone else was, including us.

My stock gradually rose during high school. Puberty finally arrived; I became a decent soccer player; I started a scandalous underground newspaper. So I've seen a good part of the popularity landscape.

I know a lot of people who were nerds in school, and they all tell the same story: there is a strong correlation between being smart and being a nerd, and an even stronger inverse correlation between being a nerd and being popular. Being smart seems to *make* you unpopular.

Why? To someone in school now, that may seem an odd question to ask. The mere fact is so overwhelming that it may seem strange to imagine that it could be any other way. But it could. Being smart doesn't make you an outcast in elementary school. Nor does it harm you in the real world. Nor, as far as I can tell, is the problem so bad in most other countries. But in a typical American secondary school, being smart is likely to make your life difficult. Why?

The key to this mystery is to rephrase the question slightly. Why don't smart kids

make themselves popular? If they're so smart, why don't they figure out how popularity works and beat the system, just as they do for standardized tests?

One argument says that this would be impossible, that the smart kids are unpopular because the other kids envy them for being smart, and nothing they could do could make them popular. I wish. If the other kids in junior high school envied me, they did a great job of concealing it. And in any case, if being smart were really an enviable quality, the girls would have broken ranks. The guys that guys envy, girls like.

In the schools I went to, being smart just didn't matter much. Kids didn't admire it or despise it. All other things being equal, they would have preferred to be on the smart side of average rather than the dumb side, but intelligence counted far less than, say, physical appearance, charisma, or athletic ability.

So if intelligence in itself is not a factor in popularity, why are smart kids so consistently unpopular? The answer, I think, is that they don't really want to be popular.

If someone had told me that at the time, I would have laughed at him. Being unpopular in school makes kids miserable, some of them so miserable that they commit suicide. Telling me that I didn't want to be popular would have seemed like telling someone dying of thirst in a desert that he didn't want a glass of water. Of course I wanted to be popular.

But in fact I didn't, not enough. There was something else I wanted more: to be smart. Not simply to do well in school, though that counted for something, but to design beautiful rockets, or to write well, or to understand how to program computers. In general, to make great things.

At the time I never tried to separate my wants and weigh them against one another. If I had, I would have seen that being smart was more important. If someone had offered me the chance to be the most popular kid in school, but only at the price of being of average intelligence (humor me here), I wouldn't have taken it.

Much as they suffer from their unpopularity, I don't think many nerds would. To them the thought of average intelligence is unbearable. But most kids would take that deal. For half of them, it would be a step up. Even for someone in the eightieth percentile (assuming, as everyone seemed to then, that intelligence is a scalar), who wouldn't drop thirty points in exchange for being loved and admired by everyone?

And that, I think, is the root of the problem. Nerds serve two masters. They want to be popular, certainly, but they want even more to be smart. And popularity is not something you can do in your spare time, not in the fiercely competitive environment of an American secondary school.

Alberti, arguably the archetype of the Renaissance Man, writes that "no art, however minor, demands less than total dedication if you want to excel in it." I wonder if anyone in the world works harder at anything than American school kids work at popularity. Navy SEALs and neurosurgery residents seem slackers by comparison. They occasionally take vacations; some even have hobbies. An American teenager may work at being popular every waking hour, 365 days a year.

I don't mean to suggest they do this consciously. Some of them truly are little Machiavellis, but what I really mean here is that teenagers are always on duty as conformists.

For example, teenage kids pay a great deal of attention to clothes. They don't consciously dress to be popular. They dress to look good. But to who? To the other kids. Other kids' opinions become their definition of right, not just for clothes, but for almost everything they do, right down to the way they walk. And so every effort they make to do things "right" is also, consciously or not, an effort to be more popular.

Nerds don't realize this. They don't realize that it takes work to be popular. In general, people outside some very demanding field don't realize the extent to which success depends on constant (though often unconscious) effort. For example, most people seem to consider the ability to draw as some kind of innate quality, like being tall. In fact, most people who "can draw" like drawing, and have spent many hours doing it; that's why they're good at it. Likewise, popular isn't just something you are or you aren't, but something you make yourself.

The main reason nerds are unpopular is that they have other things to think about. Their attention is drawn to books or the natural world, not fashions and parties. They're like someone trying to play soccer while balancing a glass of water on his head. Other players who can focus their whole attention on the game beat them effortlessly, and wonder why they seem so incapable.

Even if nerds cared as much as other kids about popularity, being popular would be more work for them. The popular kids learned to be popular, and to want to be popular, the same way the nerds learned to be smart, and to want to be smart: from their parents. While the nerds were being trained to get the right answers, the popular kids were being trained to please.

So far I've been finessing the relationship between smart and nerd, using them as if they were interchangeable. In fact it's only the context that makes them so. A nerd is someone who isn't socially adept enough. But "enough" depends on where you are. In a typical American school, standards for coolness are so high (or at least, so specific) that you don't have to be especially awkward to look awkward by comparison.

Few smart kids can spare the attention that popularity requires. Unless they also happen to be good-looking, natural athletes, or siblings of popular kids, they'll tend to become nerds. And that's why smart people's lives are worst between, say, the ages of eleven and seventeen. Life at that age revolves far more around popularity than before or after.

Before that, kids' lives are dominated by their parents, not by other kids. Kids do care what their peers think in elementary school, but this isn't their whole life, as it later becomes.

Around the age of eleven, though, kids seem to start treating their family as a day job. They create a new world among themselves, and standing in this world is what matters, not standing in their family. Indeed, being in trouble in their family can win them points in the world they care about.

The problem is, the world these kids create for themselves is at first a very crude one. If you leave a bunch of eleven-year-olds to their own devices, what you get is *Lord of the Flies.* Like a lot of American kids, I read this book in school. Presumably it was not a coincidence. Presumably someone wanted to point out to us that we were savages, and that we had made ourselves a cruel and stupid world. This was too subtle for me. While the book seemed entirely believable, I didn't get the additional message. I wish they had just told us outright that we were savages and our world was stupid.

Nerds would find their unpopularity more bearable if it merely caused them to be ignored. Unfortunately, to be unpopular in school is to be actively persecuted.

Why? Once again, anyone currently in school might think this a strange question to ask. How could things be any other way? But they could be. Adults don't normally persecute nerds. Why do teenage kids do it?

Partly because teenagers are still half children, and many children are just intrinsically cruel. Some torture nerds for the same reason they pull the legs off spiders. Before you develop a conscience, torture is amusing.

Another reason kids persecute nerds is to make themselves feel better. When you tread water, you lift yourself up by pushing water down. Likewise, in any social hierarchy, people unsure of their own position will try to emphasize it by maltreating those they think rank below. I've read that this is why poor whites in the United States are the group most hostile to blacks.

But I think the main reason other kids persecute nerds is that it's part of the mechanism of popularity. Popularity is only partially about individual attractiveness. It's much more about alliances. To become more popular, you need to be constantly doing things that bring you close to other popular people, and

nothing brings people closer than a common enemy.

Like a politician who wants to distract voters from bad times at home, you can create an enemy if there isn't a real one. By singling out and persecuting a nerd, a group of kids from higher in the hierarchy create bonds between themselves. Attacking an outsider makes them all insiders. This is why the worst cases of bullying happen with groups. Ask any nerd: you get much worse treatment from a group of kids than from any individual bully, however sadistic.

If it's any consolation to the nerds, it's nothing personal. The group of kids who band together to pick on you are doing the same thing, and for the same reason, as a bunch of guys who get together to go hunting. They don't actually hate you. They just need something to chase.

Because they're at the bottom of the scale, nerds are a safe target for the entire school. If I remember correctly, the most popular kids don't persecute nerds; they don't need to stoop to such things. Most of the persecution comes from kids lower down, the nervous middle classes.

The trouble is, there are a lot of them. The distribution of popularity is not a pyramid, but tapers at the bottom like a pear. The least popular group is quite small. (I believe we were the only D table in our cafeteria map.) So there are more people who want to pick on nerds than there are nerds.

As well as gaining points by distancing oneself from unpopular kids, one loses points by being close to them. A woman I know says that in high school she liked nerds, but was afraid to be seen talking to them because the other girls would make fun of her. Unpopularity is a communicable disease; kids too nice to pick on nerds will still ostracize them in self-defense.

It's no wonder, then, that smart kids tend to be unhappy in middle school and high school. Their other interests leave them little attention to spare for popularity, and since popularity resembles a zero-sum game, this in turn makes them targets for the whole school. And the strange thing is, this nightmare scenario happens without any conscious malice, merely because of the shape of the situation.

For me the worst stretch was junior high, when kid culture was new and harsh, and the specialization that would later gradually separate the smarter kids had barely begun. Nearly everyone I've talked to agrees: the nadir is somewhere between eleven and fourteen.

In our school it was eighth grade, which was ages twelve and thirteen for me. There was a brief sensation that year when one of our teachers overheard a group of girls waiting for the school bus, and was so shocked that the next day she devoted the whole class to an eloquent plea not to be so cruel to one another.

It didn't have any noticeable effect. What struck me at the time was that she was surprised. You mean she doesn't know the kind of things they say to one another? You mean this isn't normal?

It's important to realize that, no, the adults don't know what the kids are doing to one another. They know, in the abstract, that kids are monstrously cruel to one another, just as we know in the abstract that people get tortured in poorer countries. But, like us, they don't like to dwell on this depressing fact, and they don't see evidence of specific abuses unless they go looking for it.

Public school teachers are in much the same position as prison wardens. Wardens' main concern is to keep the prisoners on the premises. They also need to keep them fed, and as far as possible prevent them from killing one another. Beyond that, they want to have as little to do with the prisoners as possible, so they leave them to create whatever social organization they want. From what I've read, the society that the prisoners create is warped, savage, and pervasive, and it is no fun to be at the bottom of it.

In outline, it was the same at the schools I went to. The most important thing was to stay on the premises. While there, the authorities fed you, prevented overt violence, and made some effort to teach you something. But beyond that they didn't want to have too much to do with the kids. Like prison wardens, the teachers mostly left us to ourselves. And, like prisoners, the culture we created was barbaric.

Why is the real world more hospitable to nerds? It might seem that the answer is simply that it's populated by adults, who are too mature to pick on one another. But I don't think this is true. Adults in prison certainly pick on one another. And so, apparently, do society wives; in some parts of Manhattan, life for women sounds like a continuation of high school, with all the same petty intrigues.

I think the important thing about the real world is not that it's populated by adults, but that it's very large, and the things you do have real effects. That's what school, prison, and ladies-who-lunch all lack. The inhabitants of all those worlds are trapped in little bubbles where nothing they do can have more than a local effect. Naturally these societies degenerate into savagery. They have no function for their form to follow.

When the things you do have real effects, it's no longer enough just to be pleasing. It starts to be important to get the right answers, and that's where nerds show to advantage. Bill Gates will of course come to mind. Though notoriously lacking in social skills, he gets the right answers, at least as measured in revenue.

The other thing that's different about the real world is that it's much larger. In a large enough pool, even the smallest minorities can achieve a critical mass if they clump together. Out in the real world, nerds collect in certain places and form their

own societies where intelligence is the most important thing. Sometimes the current even starts to flow in the other direction: sometimes, particularly in university math and science departments, nerds deliberately exaggerate their awkwardness in order to seem smarter. John Nash so admired Norbert Wiener that he adopted his habit of touching the wall as he walked down a corridor.

As a thirteen-year-old kid, I didn't have much more experience of the world than what I saw immediately around me. The warped little world we lived in was, I thought, *the world.* The world seemed cruel and boring, and I'm not sure which was worse.

Because I didn't fit into this world, I thought that something must be wrong with me. I didn't realize that the reason we nerds didn't fit in was that in some ways we were a step ahead. We were already thinking about the kind of things that matter in the real world, instead of spending all our time playing an exacting but mostly pointless game like the others.

We were a bit like an adult would be if he were thrust back into middle school. He wouldn't know the right clothes to wear, the right music to like, the right slang to use. He'd seem to the kids a complete alien. The thing is, he'd know enough not to care what they thought. We had no such confidence.

A lot of people seem to think it's good for smart kids to be thrown together with "normal" kids at this stage of their lives. Perhaps. But in at least some cases the reason the nerds don't fit in really is that everyone else is crazy. I remember sitting in the audience at a "pep rally" at my high school, watching as the cheerleaders threw an effigy of an opposing player into the audience to be torn to pieces. I felt like an explorer witnessing some bizarre tribal ritual.

If I could go back and give my thirteen year old self some advice, the main thing I'd tell him would be to stick his head up and look around. I didn't really grasp it at the time, but the whole world we lived in was as fake as a Twinkie. Not just school, but the entire town. Why do people move to suburbia? To have kids! So no wonder it seemed boring and sterile. The whole place was a giant nursery, an artificial town created explicitly for the purpose of breeding children.

Where I grew up, it felt as if there was nowhere to go, and nothing to do. This was no accident. Suburbs are deliberately designed to exclude the outside world, because it contains things that could endanger children.

And as for the schools, they were just holding pens within this fake world. Officially the purpose of schools is to teach kids. In fact their primary purpose is to keep kids locked up in one place for a big chunk of the day so adults can get things done. And I have no problem with this: in a specialized industrial society, it would

be a disaster to have kids running around loose.

What bothers me is not that the kids are kept in prisons, but that (a) they aren't told about it, and (b) the prisons are run mostly by the inmates. Kids are sent off to spend six years memorizing meaningless facts in a world ruled by a caste of giants who run after an oblong brown ball, as if this were the most natural thing in the world. And if they balk at this surreal cocktail, they're called misfits.

Life in this twisted world is stressful for the kids. And not just for the nerds. Like any war, it's damaging even to the winners.

Adults can't avoid seeing that teenage kids are tormented. So why don't they do something about it? Because they blame it on puberty. The reason kids are so unhappy, adults tell themselves, is that monstrous new chemicals, *hormones*, are now coursing through their bloodstream and messing up everything. There's nothing wrong with the system; it's just inevitable that kids will be miserable at that age.

This idea is so pervasive that even the kids believe it, which probably doesn't help. Someone who thinks his feet naturally hurt is not going to stop to consider the possibility that he is wearing the wrong size shoes.

I'm suspicious of this theory that thirteen-year-old kids are intrinsically messed up. If it's physiological, it should be universal. Are Mongol nomads all nihilists at thirteen? I've read a lot of history, and I have not seen a single reference to this supposedly universal fact before the twentieth century. Teenage apprentices in the Renaissance seem to have been cheerful and eager. They got in fights and played tricks on one another of course (Michelangelo had his nose broken by a bully), but they weren't crazy.

As far as I can tell, the concept of the hormone-crazed teenager is coeval with suburbia. I don't think this is a coincidence. I think teenagers are driven crazy by the life they're made to lead. Teenage apprentices in the Renaissance were working dogs. Teenagers now are neurotic lapdogs. Their craziness is the craziness of the idle everywhere.

When I was in school, suicide was a constant topic among the smarter kids. No one I knew did it, but several planned to, and some may have tried. Mostly this was just a pose. Like other teenagers, we loved the dramatic, and suicide seemed very dramatic. But partly it was because our lives were at times genuinely miserable.

Bullying was only part of the problem. Another problem, and possibly an even worse one, was that we never had anything real to work on. Humans like to work;

in most of the world, your work is your identity. And all the work we did was [pointless](), or seemed so at the time.

At best it was practice for real work we might do far in the future, so far that we didn't even know at the time what we were practicing for. More often it was just an arbitrary series of hoops to jump through, words without content designed mainly for testability. (The three main causes of the Civil War were.... Test: List the three main causes of the Civil War.)

And there was no way to opt out. The adults had agreed among themselves that this was to be the route to college. The only way to escape this empty life was to submit to it.

Teenage kids used to have a more active role in society. In pre-industrial times, they were all apprentices of one sort or another, whether in shops or on farms or even on warships. They weren't left to create their own societies. They were junior members of adult societies.

Teenagers seem to have respected adults more then, because the adults were the visible experts in the skills they were trying to learn. Now most kids have little idea what their parents do in their distant offices, and see no connection (indeed, there is precious little) between schoolwork and the work they'll do as adults.

And if teenagers respected adults more, adults also had more use for teenagers. After a couple years' training, an apprentice could be a real help. Even the newest apprentice could be made to carry messages or sweep the workshop.

Now adults have no immediate use for teenagers. They would be in the way in an office. So they drop them off at school on their way to work, much as they might drop the dog off at a kennel if they were going away for the weekend.

What happened? We're up against a hard one here. The cause of this problem is the same as the cause of so many present ills: specialization. As jobs become more specialized, we have to train longer for them. Kids in pre-industrial times started working at about 14 at the latest; kids on farms, where most people lived, began far earlier. Now kids who go to college don't start working full-time till 21 or 22. With some degrees, like MDs and PhDs, you may not finish your training till 30.

Teenagers now are useless, except as cheap labor in industries like fast food, which evolved to exploit precisely this fact. In almost any other kind of work, they'd be a net loss. But they're also too young to be left unsupervised. Someone has to watch over them, and the most efficient way to do this is to collect them together in one place. Then a few adults can watch all of them.

If you stop there, what you're describing is literally a prison, albeit a part-time one. The problem is, many schools practically do stop there. The stated purpose of

schools is to educate the kids. But there is no external pressure to do this well. And so most schools do such a bad job of teaching that the kids don't really take it seriously-- not even the smart kids. Much of the time we were all, students and teachers both, just going through the motions.

In my high school French class we were supposed to read Hugo's *Les Miserables.* I don't think any of us knew French well enough to make our way through this enormous book. Like the rest of the class, I just skimmed the Cliff's Notes. When we were given a test on the book, I noticed that the questions sounded odd. They were full of long words that our teacher wouldn't have used. Where had these questions come from? From the Cliff's Notes, it turned out. The teacher was using them too. We were all just pretending.

There are certainly great public school teachers. The energy and imagination of my fourth grade teacher, Mr. Mihalko, made that year something his students still talk about, thirty years later. But teachers like him were individuals swimming upstream. They couldn't fix the system.

In almost any group of people you'll find hierarchy. When groups of adults form in the real world, it's generally for some common purpose, and the leaders end up being those who are best at it. The problem with most schools is, they have no purpose. But hierarchy there must be. And so the kids make one out of nothing.

We have a phrase to describe what happens when rankings have to be created without any meaningful criteria. We say that the situation *degenerates into a popularity contest.* And that's exactly what happens in most American schools. Instead of depending on some real test, one's rank depends mostly on one's ability to increase one's rank. It's like the court of Louis XIV. There is no external opponent, so the kids become one another's opponents.

When there is some real external test of skill, it isn't painful to be at the bottom of the hierarchy. A rookie on a football team doesn't resent the skill of the veteran; he hopes to be like him one day and is happy to have the chance to learn from him. The veteran may in turn feel a sense of *noblesse oblige*. And most importantly, their status depends on how well they do against opponents, not on whether they can push the other down.

Court hierarchies are another thing entirely. This type of society debases anyone who enters it. There is neither admiration at the bottom, nor *noblesse oblige* at the top. It's kill or be killed.

This is the sort of society that gets created in American secondary schools. And it happens because these schools have no real purpose beyond keeping the kids all in one place for a certain number of hours each day. What I didn't realize at the time, and in fact didn't realize till very recently, is that the twin horrors of school life, the cruelty and the boredom, both have the same cause.

The mediocrity of American public schools has worse consequences than just making kids unhappy for six years. It breeds a rebelliousness that actively drives kids away from the things they're supposed to be learning.

Like many nerds, probably, it was years after high school before I could bring myself to read anything we'd been assigned then. And I lost more than books. I mistrusted words like "character" and "integrity" because they had been so debased by adults. As they were used then, these words all seemed to mean the same thing: obedience. The kids who got praised for these qualities tended to be at best dull-witted prize bulls, and at worst facile schmoozers. If that was what character and integrity were, I wanted no part of them.

The word I most misunderstood was "tact." As used by adults, it seemed to mean keeping your mouth shut. I assumed it was derived from the same root as "tacit" and "taciturn," and that it literally meant being quiet. I vowed that I would never be tactful; they were never going to shut me up. In fact, it's derived from the same root as "tactile," and what it means is to have a deft touch. Tactful is the opposite of clumsy. I don't think I learned this until college.

Nerds aren't the only losers in the popularity rat race. Nerds are unpopular because they're distracted. There are other kids who deliberately opt out because they're so disgusted with the whole process.

Teenage kids, even rebels, don't like to be alone, so when kids opt out of the system, they tend to do it as a group. At the schools I went to, the focus of rebellion was drug use, specifically marijuana. The kids in this tribe wore black concert t-shirts and were called "freaks."

Freaks and nerds were allies, and there was a good deal of overlap between them. Freaks were on the whole smarter than other kids, though never studying (or at least never appearing to) was an important tribal value. I was more in the nerd camp, but I was friends with a lot of freaks.

They used drugs, at least at first, for the social bonds they created. It was something to do together, and because the drugs were illegal, it was a shared badge of rebellion.

I'm not claiming that bad schools are the whole reason kids get into trouble with drugs. After a while, drugs have their own momentum. No doubt some of the freaks ultimately used drugs to escape from other problems-- trouble at home, for example. But, in my school at least, the reason most kids *started* using drugs was rebellion. Fourteen-year-olds didn't start smoking pot because they'd heard it would help them forget their problems. They started because they wanted to join a

different tribe.

Misrule breeds rebellion; this is not a new idea. And yet the authorities still for the most part act as if drugs were themselves the cause of the problem.

The real problem is the emptiness of school life. We won't see solutions till adults realize that. The adults who may realize it first are the ones who were themselves nerds in school. Do you want your kids to be as unhappy in eighth grade as you were? I wouldn't. Well, then, is there anything we can do to fix things? Almost certainly. There is nothing inevitable about the current system. It has come about mostly by default.

Adults, though, are busy. Showing up for school plays is one thing. Taking on the educational bureaucracy is another. Perhaps a few will have the energy to try to change things. I suspect the hardest part is realizing that you can.

Nerds still in school should not hold their breath. Maybe one day a heavily armed force of adults will show up in helicopters to rescue you, but they probably won't be coming this month. Any immediate improvement in nerds' lives is probably going to have to come from the nerds themselves.

Merely understanding the situation they're in should make it less painful. Nerds aren't losers. They're just playing a different game, and a game much closer to the one played in the real world. Adults know this. It's hard to find successful adults now who don't claim to have been nerds in high school.

It's important for nerds to realize, too, that school is not life. School is a strange, artificial thing, half sterile and half feral. It's all-encompassing, like life, but it isn't the real thing. It's only temporary, and if you look, you can see beyond it even while you're still in it.

If life seems awful to kids, it's neither because hormones are turning you all into monsters (as your parents believe), nor because life actually is awful (as you believe). It's because the adults, who no longer have any economic use for you, have abandoned you to spend years cooped up together with nothing real to do. *Any* society of that type is awful to live in. You don't have to look any further to explain why teenage kids are unhappy.

I've said some harsh things in this essay, but really the thesis is an optimistic one-- that several problems we take for granted are in fact not insoluble after all. Teenage kids are not inherently unhappy monsters. That should be encouraging news to kids and adults both.

Weicker for reading drafts of this essay, and Maria Daniels for scanning photos.

•

[Re: Why Nerds are Unpopular](#)

•

[Gateway High School, 1981](#)

•

[Japanese Translation](#)

•

[French Translation](#)

•

[My War With Brian](#)

•

[Buttons](#)

•

[Portuguese Translation](#)

•

[Spanish Translation](#)

The Hundred-Year Language

April 2003

(This essay is derived from a keynote talk at PyCon 2003.)

It's hard to predict what life will be like in a hundred years. There are only a few things we can say with certainty. We know that everyone will drive flying cars, that zoning laws will be relaxed to allow buildings hundreds of stories tall, that it will be dark most of the time, and that women will all be trained in the martial arts. Here I want to zoom in on one detail of this picture. What kind of programming language will they use to write the software controlling those flying cars?

This is worth thinking about not so much because we'll actually get to use these languages as because, if we're lucky, we'll use languages on the path from this point to that.

I think that, like species, languages will form evolutionary trees, with dead-ends branching off all over. We can see this happening already. Cobol, for all its sometime popularity, does not seem to have any intellectual descendants. It is an evolutionary dead-end-- a Neanderthal language.

I predict a similar fate for Java. People sometimes send me mail saying, "How can you say that Java won't turn out to be a successful language? It's already a successful language." And I admit that it is, if you measure success by shelf space taken up by books on it (particularly individual books on it), or by the number of undergrads who believe they have to learn it to get a job. When I say Java won't turn out to be a successful language, I mean something more specific: that Java will turn out to be an evolutionary dead-end, like Cobol.

This is just a guess. I may be wrong. My point here is not to dis Java, but to raise

the issue of evolutionary trees and get people asking, where on the tree is language X? The reason to ask this question isn't just so that our ghosts can say, in a hundred years, I told you so. It's because staying close to the main branches is a useful heuristic for finding languages that will be good to program in now.

At any given time, you're probably happiest on the main branches of an evolutionary tree. Even when there were still plenty of Neanderthals, it must have sucked to be one. The Cro-Magnons would have been constantly coming over and beating you up and stealing your food.

The reason I want to know what languages will be like in a hundred years is so that I know what branch of the tree to bet on now.

The evolution of languages differs from the evolution of species because branches can converge. The Fortran branch, for example, seems to be merging with the descendants of Algol. In theory this is possible for species too, but it's not likely to have happened to any bigger than a cell.

Convergence is more likely for languages partly because the space of possibilities is smaller, and partly because mutations are not random. Language designers deliberately incorporate ideas from other languages.

It's especially useful for language designers to think about where the evolution of programming languages is likely to lead, because they can steer accordingly. In that case, "stay on a main branch" becomes more than a way to choose a good language. It becomes a heuristic for making the right decisions about language design.

Any programming language can be divided into two parts: some set of fundamental operators that play the role of axioms, and the rest of the language, which could in principle be written in terms of these fundamental operators.

I think the fundamental operators are the most important factor in a language's long term survival. The rest you can change. It's like the rule that in buying a house you should consider location first of all. Everything else you can fix later, but you can't fix the location.

I think it's important not just that the axioms be well chosen, but that there be few of them. Mathematicians have always felt this way about axioms-- the fewer, the better-- and I think they're onto something.

At the very least, it has to be a useful exercise to look closely at the core of a language to see if there are any axioms that could be weeded out. I've found in my long career as a slob that cruft breeds cruft, and I've seen this happen in software

as well as under beds and in the corners of rooms.

I have a hunch that the main branches of the evolutionary tree pass through the languages that have the smallest, cleanest cores. The more of a language you can write in itself, the better.

Of course, I'm making a big assumption in even asking what programming languages will be like in a hundred years. Will we even be writing programs in a hundred years? Won't we just tell computers what we want them to do?

There hasn't been a lot of progress in that department so far. My guess is that a hundred years from now people will still tell computers what to do using programs we would recognize as such. There may be tasks that we solve now by writing programs and which in a hundred years you won't have to write programs to solve, but I think there will still be a good deal of programming of the type that we do today.

It may seem presumptuous to think anyone can predict what any technology will look like in a hundred years. But remember that we already have almost fifty years of history behind us. Looking forward a hundred years is a graspable idea when we consider how slowly languages have evolved in the past fifty.

Languages evolve slowly because they're not really technologies. Languages are notation. A program is a formal description of the problem you want a computer to solve for you. So the rate of evolution in programming languages is more like the rate of evolution in mathematical notation than, say, transportation or communications. Mathematical notation does evolve, but not with the giant leaps you see in technology.

Whatever computers are made of in a hundred years, it seems safe to predict they will be much faster than they are now. If Moore's Law continues to put out, they will be 74 quintillion (73,786,976,294,838,206,464) times faster. That's kind of hard to imagine. And indeed, the most likely prediction in the speed department may be that Moore's Law will stop working. Anything that is supposed to double every eighteen months seems likely to run up against some kind of fundamental limit eventually. But I have no trouble believing that computers will be very much faster. Even if they only end up being a paltry million times faster, that should change the ground rules for programming languages substantially. Among other things, there will be more room for what would now be considered slow languages, meaning languages that don't yield very efficient code.

And yet some applications will still demand speed. Some of the problems we want to solve with computers are created by computers; for example, the rate at which you have to process video images depends on the rate at which another computer

can generate them. And there is another class of problems which inherently have an unlimited capacity to soak up cycles: image rendering, cryptography, simulations.

If some applications can be increasingly inefficient while others continue to demand all the speed the hardware can deliver, faster computers will mean that languages have to cover an ever wider range of efficiencies. We've seen this happening already. Current implementations of some popular new languages are shockingly wasteful by the standards of previous decades.

This isn't just something that happens with programming languages. It's a general historical trend. As technologies improve, each generation can do things that the previous generation would have considered wasteful. People thirty years ago would be astonished at how casually we make long distance phone calls. People a hundred years ago would be even more astonished that a package would one day travel from Boston to New York via Memphis.

I can already tell you what's going to happen to all those extra cycles that faster hardware is going to give us in the next hundred years. They're nearly all going to be wasted.

I learned to program when computer power was scarce. I can remember taking all the spaces out of my Basic programs so they would fit into the memory of a 4K TRS-80. The thought of all this stupendously inefficient software burning up cycles doing the same thing over and over seems kind of gross to me. But I think my intuitions here are wrong. I'm like someone who grew up poor, and can't bear to spend money even for something important, like going to the doctor.

Some kinds of waste really are disgusting. SUVs, for example, would arguably be gross even if they ran on a fuel which would never run out and generated no pollution. SUVs are gross because they're the solution to a gross problem. (How to make minivans look more masculine.) But not all waste is bad. Now that we have the infrastructure to support it, counting the minutes of your long-distance calls starts to seem niggling. If you have the resources, it's more elegant to think of all phone calls as one kind of thing, no matter where the other person is.

There's good waste, and bad waste. I'm interested in good waste-- the kind where, by spending more, we can get simpler designs. How will we take advantage of the opportunities to waste cycles that we'll get from new, faster hardware?

The desire for speed is so deeply engrained in us, with our puny computers, that it will take a conscious effort to overcome it. In language design, we should be consciously seeking out situations where we can trade efficiency for even the smallest increase in convenience.

Most data structures exist because of speed. For example, many languages today have both strings and lists. Semantically, strings are more or less a subset of lists in which the elements are characters. So why do you need a separate data type? You don't, really. Strings only exist for efficiency. But it's lame to clutter up the semantics of the language with hacks to make programs run faster. Having strings in a language seems to be a case of premature optimization.

If we think of the core of a language as a set of axioms, surely it's gross to have additional axioms that add no expressive power, simply for the sake of efficiency. Efficiency is important, but I don't think that's the right way to get it.

The right way to solve that problem, I think, is to separate the meaning of a program from the implementation details. Instead of having both lists and strings, have just lists, with some way to give the compiler optimization advice that will allow it to lay out strings as contiguous bytes if necessary.

Since speed doesn't matter in most of a program, you won't ordinarily need to bother with this sort of micromanagement. This will be more and more true as computers get faster.

Saying less about implementation should also make programs more flexible. Specifications change while a program is being written, and this is not only inevitable, but desirable.

The word "essay" comes from the French verb "essayer", which means "to try". An essay, in the original sense, is something you write to try to figure something out. This happens in software too. I think some of the best programs were essays, in the sense that the authors didn't know when they started exactly what they were trying to write.

Lisp hackers already know about the value of being flexible with data structures. We tend to write the first version of a program so that it does everything with lists. These initial versions can be so shockingly inefficient that it takes a conscious effort not to think about what they're doing, just as, for me at least, eating a steak requires a conscious effort not to think where it came from.

What programmers in a hundred years will be looking for, most of all, is a language where you can throw together an unbelievably inefficient version 1 of a program with the least possible effort. At least, that's how we'd describe it in present-day terms. What they'll say is that they want a language that's easy to program in.

Inefficient software isn't gross. What's gross is a language that makes programmers do needless work. Wasting programmer time is the true inefficiency, not wasting machine time. This will become ever more clear as computers get faster.

I think getting rid of strings is already something we could bear to think about. We did it in Arc, and it seems to be a win; some operations that would be awkward to describe as regular expressions can be described easily as recursive functions.

How far will this flattening of data structures go? I can think of possibilities that shock even me, with my conscientiously broadened mind. Will we get rid of arrays, for example? After all, they're just a subset of hash tables where the keys are vectors of integers. Will we replace hash tables themselves with lists?

There are more shocking prospects even than that. The Lisp that McCarthy described in 1960, for example, didn't have numbers. Logically, you don't need to have a separate notion of numbers, because you can represent them as lists: the integer n could be represented as a list of n elements. You can do math this way. It's just unbearably inefficient.

No one actually proposed implementing numbers as lists in practice. In fact, McCarthy's 1960 paper was not, at the time, intended to be implemented at all. It was a theoretical exercise, an attempt to create a more elegant alternative to the Turing Machine. When someone did, unexpectedly, take this paper and translate it into a working Lisp interpreter, numbers certainly weren't represented as lists; they were represented in binary, as in every other language.

Could a programming language go so far as to get rid of numbers as a fundamental data type? I ask this not so much as a serious question as as a way to play chicken with the future. It's like the hypothetical case of an irresistible force meeting an immovable object-- here, an unimaginably inefficient implementation meeting unimaginably great resources. I don't see why not. The future is pretty long. If there's something we can do to decrease the number of axioms in the core language, that would seem to be the side to bet on as t approaches infinity. If the idea still seems unbearable in a hundred years, maybe it won't in a thousand.

Just to be clear about this, I'm not proposing that all numerical calculations would actually be carried out using lists. I'm proposing that the core language, prior to any additional notations about implementation, be defined this way. In practice any program that wanted to do any amount of math would probably represent numbers in binary, but this would be an optimization, not part of the core language semantics.

Another way to burn up cycles is to have many layers of software between the application and the hardware. This too is a trend we see happening already: many recent languages are compiled into byte code. Bill Woods once told me that, as a rule of thumb, each layer of interpretation costs a factor of 10 in speed. This extra cost buys you flexibility.

The very first version of Arc was an extreme case of this sort of multi-level slowness, with corresponding benefits. It was a classic "metacircular" interpreter written on top of Common Lisp, with a definite family resemblance to the eval function defined in McCarthy's original Lisp paper. The whole thing was only a couple hundred lines of code, so it was very easy to understand and change. The Common Lisp we used, CLisp, itself runs on top of a byte code interpreter. So here we had two levels of interpretation, one of them (the top one) shockingly inefficient, and the language was usable. Barely usable, I admit, but usable.

Writing software as multiple layers is a powerful technique even within applications. Bottom-up programming means writing a program as a series of layers, each of which serves as a language for the one above. This approach tends to yield smaller, more flexible programs. It's also the best route to that holy grail, reusability. A language is by definition reusable. The more of your application you can push down into a language for writing that type of application, the more of your software will be reusable.

Somehow the idea of reusability got attached to object-oriented programming in the 1980s, and no amount of evidence to the contrary seems to be able to shake it free. But although some object-oriented software is reusable, what makes it reusable is its bottom-upness, not its object-orientedness. Consider libraries: they're reusable because they're language, whether they're written in an object-oriented style or not.

I don't predict the demise of object-oriented programming, by the way. Though I don't think it has much to offer good programmers, except in certain specialized domains, it is irresistible to large organizations. Object-oriented programming offers a sustainable way to write spaghetti code. It lets you accrete programs as a series of patches. Large organizations always tend to develop software this way, and I expect this to be as true in a hundred years as it is today.

As long as we're talking about the future, we had better talk about parallel computation, because that's where this idea seems to live. That is, no matter when you're talking, parallel computation seems to be something that is going to happen in the future.

Will the future ever catch up with it? People have been talking about parallel computation as something imminent for at least 20 years, and it hasn't affected programming practice much so far. Or hasn't it? Already chip designers have to think about it, and so must people trying to write systems software on multi-cpu computers.

The real question is, how far up the ladder of abstraction will parallelism go? In a hundred years will it affect even application programmers? Or will it be something that compiler writers think about, but which is usually invisible in the source code

of applications?

One thing that does seem likely is that most opportunities for parallelism will be wasted. This is a special case of my more general prediction that most of the extra computer power we're given will go to waste. I expect that, as with the stupendous speed of the underlying hardware, parallelism will be something that is available if you ask for it explicitly, but ordinarily not used. This implies that the kind of parallelism we have in a hundred years will not, except in special applications, be massive parallelism. I expect for ordinary programmers it will be more like being able to fork off processes that all end up running in parallel.

And this will, like asking for specific implementations of data structures, be something that you do fairly late in the life of a program, when you try to optimize it. Version 1s will ordinarily ignore any advantages to be got from parallel computation, just as they will ignore advantages to be got from specific representations of data.

Except in special kinds of applications, parallelism won't pervade the programs that are written in a hundred years. It would be premature optimization if it did.

How many programming languages will there be in a hundred years? There seem to be a huge number of new programming languages lately. Part of the reason is that faster hardware has allowed programmers to make different tradeoffs between speed and convenience, depending on the application. If this is a real trend, the hardware we'll have in a hundred years should only increase it.

And yet there may be only a few widely-used languages in a hundred years. Part of the reason I say this is optimism: it seems that, if you did a really good job, you could make a language that was ideal for writing a slow version 1, and yet with the right optimization advice to the compiler, would also yield very fast code when necessary. So, since I'm optimistic, I'm going to predict that despite the huge gap they'll have between acceptable and maximal efficiency, programmers in a hundred years will have languages that can span most of it.

As this gap widens, profilers will become increasingly important. Little attention is paid to profiling now. Many people still seem to believe that the way to get fast applications is to write compilers that generate fast code. As the gap between acceptable and maximal performance widens, it will become increasingly clear that the way to get fast applications is to have a good guide from one to the other.

When I say there may only be a few languages, I'm not including domain-specific "little languages". I think such embedded languages are a great idea, and I expect them to proliferate. But I expect them to be written as thin enough skins that users can see the general-purpose language underneath.

Who will design the languages of the future? One of the most exciting trends in the last ten years has been the rise of open-source languages like Perl, Python, and Ruby. Language design is being taken over by hackers. The results so far are messy, but encouraging. There are some stunningly novel ideas in Perl, for example. Many are stunningly bad, but that's always true of ambitious efforts. At its current rate of mutation, God knows what Perl might evolve into in a hundred years.

It's not true that those who can't do, teach (some of the best hackers I know are professors), but it is true that there are a lot of things that those who teach can't do. Research imposes constraining caste restrictions. In any academic field there are topics that are ok to work on and others that aren't. Unfortunately the distinction between acceptable and forbidden topics is usually based on how intellectual the work sounds when described in research papers, rather than how important it is for getting good results. The extreme case is probably literature; people studying literature rarely say anything that would be of the slightest use to those producing it.

Though the situation is better in the sciences, the overlap between the kind of work you're allowed to do and the kind of work that yields good languages is distressingly small. (Olin Shivers has grumbled eloquently about this.) For example, types seem to be an inexhaustible source of research papers, despite the fact that static typing seems to preclude true macros-- without which, in my opinion, no language is worth using.

The trend is not merely toward languages being developed as open-source projects rather than "research", but toward languages being designed by the application programmers who need to use them, rather than by compiler writers. This seems a good trend and I expect it to continue.

Unlike physics in a hundred years, which is almost necessarily impossible to predict, I think it may be possible in principle to design a language now that would appeal to users in a hundred years.

One way to design a language is to just write down the program you'd like to be able to write, regardless of whether there is a compiler that can translate it or hardware that can run it. When you do this you can assume unlimited resources. It seems like we ought to be able to imagine unlimited resources as well today as in a hundred years.

What program would one like to write? Whatever is least work. Except not quite: whatever *would be* least work if your ideas about programming weren't already influenced by the languages you're currently used to. Such influence can be so pervasive that it takes a great effort to overcome it. You'd think it would be obvious to creatures as lazy as us how to express a program with the least effort.

In fact, our ideas about what's possible tend to be so limited by whatever language we think in that easier formulations of programs seem very surprising. They're something you have to discover, not something you naturally sink into.

One helpful trick here is to use the length of the program as an approximation for how much work it is to write. Not the length in characters, of course, but the length in distinct syntactic elements-- basically, the size of the parse tree. It may not be quite true that the shortest program is the least work to write, but it's close enough that you're better off aiming for the solid target of brevity than the fuzzy, nearby one of least work. Then the algorithm for language design becomes: look at a program and ask, is there any way to write this that's shorter?

In practice, writing programs in an imaginary hundred-year language will work to varying degrees depending on how close you are to the core. Sort routines you can write now. But it would be hard to predict now what kinds of libraries might be needed in a hundred years. Presumably many libraries will be for domains that don't even exist yet. If SETI@home works, for example, we'll need libraries for communicating with aliens. Unless of course they are sufficiently advanced that they already communicate in XML.

At the other extreme, I think you might be able to design the core language today. In fact, some might argue that it was already mostly designed in 1958.

If the hundred year language were available today, would we want to program in it? One way to answer this question is to look back. If present-day programming languages had been available in 1960, would anyone have wanted to use them?

In some ways, the answer is no. Languages today assume infrastructure that didn't exist in 1960. For example, a language in which indentation is significant, like Python, would not work very well on printer terminals. But putting such problems aside-- assuming, for example, that programs were all just written on paper-- would programmers of the 1960s have liked writing programs in the languages we use now?

I think so. Some of the less imaginative ones, who had artifacts of early languages built into their ideas of what a program was, might have had trouble. (How can you manipulate data without doing pointer arithmetic? How can you implement flow charts without gotos?) But I think the smartest programmers would have had no trouble making the most of present-day languages, if they'd had them.

If we had the hundred-year language now, it would at least make a great pseudocode. What about using it to write software? Since the hundred-year language will need to generate fast code for some applications, presumably it could generate code efficient enough to run acceptably well on our hardware. We might have to give more optimization advice than users in a hundred years, but it still might be a net win.

Now we have two ideas that, if you combine them, suggest interesting possibilities: (1) the hundred-year language could, in principle, be designed today, and (2) such a language, if it existed, might be good to program in today. When you see these ideas laid out like that, it's hard not to think, why not try writing the hundred-year language now?

When you're working on language design, I think it is good to have such a target and to keep it consciously in mind. When you learn to drive, one of the principles they teach you is to align the car not by lining up the hood with the stripes painted on the road, but by aiming at some point in the distance. Even if all you care about is what happens in the next ten feet, this is the right answer. I think we can and should do the same thing with programming languages.

Notes

I believe Lisp Machine Lisp was the first language to embody the principle that declarations (except those of dynamic variables) were merely optimization advice, and would not change the meaning of a correct program. Common Lisp seems to have been the first to state this explicitly.

Thanks to Trevor Blackwell, Robert Morris, and Dan Giffin for reading drafts of this, and to Guido van Rossum, Jeremy Hylton, and the rest of the Python crew for inviting me to speak at PyCon.

You'll find this essay and 14 others in ***Hackers & Painters***.

If Lisp is So Great

May 2003

If Lisp is so great, why don't more people use it? I was asked this question by a student in the audience at a talk I gave recently. Not for the first time, either.

In languages, as in so many things, there's not much correlation between popularity and quality. Why does John Grisham (*King of Torts* sales rank, 44) outsell Jane Austen (*Pride and Prejudice* sales rank, 6191)? Would even Grisham claim that it's because he's a better writer?

Here's the first sentence of *Pride and Prejudice:*

> It is a truth universally acknowledged, that a single man in possession of a good fortune must be in want of a wife.

"It is a truth universally acknowledged?" Long words for the first sentence of a love story.

Like Jane Austen, Lisp looks hard. Its syntax, or lack of syntax, makes it look completely unlike the languages most people are used to. Before I learned Lisp, I was afraid of it too. I recently came across a notebook from 1983 in which I'd written:

> I suppose I should learn Lisp, but it seems so foreign.

Fortunately, I was 19 at the time and not too resistant to learning new things. I was so ignorant that learning almost anything meant learning new things.

People frightened by Lisp make up other reasons for not using it. The standard excuse, back when C was the default language, was that Lisp was too slow. Now that Lisp dialects are among the faster languages available, that excuse has gone away. Now the standard excuse is openly circular: that other languages are more popular.

(Beware of such reasoning. It gets you Windows.)

Popularity is always self-perpetuating, but it's especially so in programming languages. More libraries get written for popular languages, which makes them still more popular. Programs often have to work with existing programs, and this is easier if they're written in the same language, so languages spread from program

to program like a virus. And managers prefer popular languages, because they give them more leverage over developers, who can more easily be replaced.

Indeed, if programming languages were all more or less equivalent, there would be little justification for using any but the most popular. But they aren't all equivalent, not by a long shot. And that's why less popular languages, like Jane Austen's novels, continue to survive at all. When everyone else is reading the latest John Grisham novel, there will always be a few people reading Jane Austen instead.

▪

Japanese Translation

▪

Romanian Translation

▪

Spanish Translation

Hackers and Painters

May 2003

(This essay is derived from a guest lecture at Harvard, which incorporated an earlier talk at Northeastern.)

When I finished grad school in computer science I went to art school to study painting. A lot of people seemed surprised that someone interested in computers would also be interested in painting. They seemed to think that hacking and painting were very different kinds of work-- that hacking was cold, precise, and methodical, and that painting was the frenzied expression of some primal urge.

Both of these images are wrong. Hacking and painting have a lot in common. In fact, of all the different types of people I've known, hackers and painters are among the most alike.

What hackers and painters have in common is that they're both makers. Along with composers, architects, and writers, what hackers and painters are trying to do is make good things. They're not doing research per se, though if in the course of trying to make good things they discover some new technique, so much the better.

I've never liked the term "computer science." The main reason I don't like it is that there's no such thing. Computer science is a grab bag of tenuously related areas thrown together by an accident of history, like Yugoslavia. At one end you have people who are really mathematicians, but call what they're doing computer science so they can get DARPA grants. In the middle you have people working on something like the natural history of computers-- studying the behavior of algorithms for routing data through networks, for example. And then at the other extreme you have the hackers, who are trying to write interesting software, and for whom computers are just a medium of expression, as concrete is for architects or paint for painters. It's as if mathematicians, physicists, and architects all had to be in the same department.

Sometimes what the hackers do is called "software engineering," but this term is just as misleading. Good software designers are no more engineers than architects are. The border between architecture and engineering is not sharply defined, but it's there. It falls between what and how: architects decide what to do, and

engineers figure out how to do it.

What and how should not be kept too separate. You're asking for trouble if you try to decide what to do without understanding how to do it. But hacking can certainly be more than just deciding how to implement some spec. At its best, it's creating the spec-- though it turns out the best way to do that is to implement it.

Perhaps one day "computer science" will, like Yugoslavia, get broken up into its component parts. That might be a good thing. Especially if it meant independence for my native land, hacking.

Bundling all these different types of work together in one department may be convenient administratively, but it's confusing intellectually. That's the other reason I don't like the name "computer science." Arguably the people in the middle are doing something like an experimental science. But the people at either end, the hackers and the mathematicians, are not actually doing science.

The mathematicians don't seem bothered by this. They happily set to work proving theorems like the other mathematicians over in the math department, and probably soon stop noticing that the building they work in says ``computer science'' on the outside. But for the hackers this label is a problem. If what they're doing is called science, it makes them feel they ought to be acting scientific. So instead of doing what they really want to do, which is to design beautiful software, hackers in universities and research labs feel they ought to be writing research papers.

In the best case, the papers are just a formality. Hackers write cool software, and then write a paper about it, and the paper becomes a proxy for the achievement represented by the software. But often this mismatch causes problems. It's easy to drift away from building beautiful things toward building ugly things that make more suitable subjects for research papers.

Unfortunately, beautiful things don't always make the best subjects for papers. Number one, research must be original-- and as anyone who has written a PhD dissertation knows, the way to be sure that you're exploring virgin territory is to stake out a piece of ground that no one wants. Number two, research must be substantial-- and awkward systems yield meatier papers, because you can write about the obstacles you have to overcome in order to get things done. Nothing yields meaty problems like starting with the wrong assumptions. Most of AI is an example of this rule; if you assume that knowledge can be represented as a list of predicate logic expressions whose arguments represent abstract concepts, you'll have a lot of papers to write about how to make this work. As Ricky Ricardo used to say, "Lucy, you got a lot of explaining to do."

The way to create something beautiful is often to make subtle tweaks to something that already exists, or to combine existing ideas in a slightly new way. This kind of

work is hard to convey in a research paper.

So why do universities and research labs continue to judge hackers by publications? For the same reason that "scholastic aptitude" gets measured by simple-minded standardized tests, or the productivity of programmers gets measured in lines of code. These tests are easy to apply, and there is nothing so tempting as an easy test that kind of works.

Measuring what hackers are actually trying to do, designing beautiful software, would be much more difficult. You need a good sense of design to judge good design. And there is no correlation, except possibly a negative one, between people's ability to recognize good design and their confidence that they can.

The only external test is time. Over time, beautiful things tend to thrive, and ugly things tend to get discarded. Unfortunately, the amounts of time involved can be longer than human lifetimes. Samuel Johnson said it took a hundred years for a writer's reputation to converge. You have to wait for the writer's influential friends to die, and then for all their followers to die.

I think hackers just have to resign themselves to having a large random component in their reputations. In this they are no different from other makers. In fact, they're lucky by comparison. The influence of fashion is not nearly so great in hacking as it is in painting.

There are worse things than having people misunderstand your work. A worse danger is that you will yourself misunderstand your work. Related fields are where you go looking for ideas. If you find yourself in the computer science department, there is a natural temptation to believe, for example, that hacking is the applied version of what theoretical computer science is the theory of. All the time I was in graduate school I had an uncomfortable feeling in the back of my mind that I ought to know more theory, and that it was very remiss of me to have forgotten all that stuff within three weeks of the final exam.

Now I realize I was mistaken. Hackers need to understand the theory of computation about as much as painters need to understand paint chemistry. You need to know how to calculate time and space complexity and about Turing completeness. You might also want to remember at least the concept of a state machine, in case you have to write a parser or a regular expression library. Painters in fact have to remember a good deal more about paint chemistry than that.

I've found that the best sources of ideas are not the other fields that have the word "computer" in their names, but the other fields inhabited by makers. Painting has been a much richer source of ideas than the theory of computation.

For example, I was taught in college that one ought to figure out a program completely on paper before even going near a computer. I found that I did not program this way. I found that I liked to program sitting in front of a computer, not a piece of paper. Worse still, instead of patiently writing out a complete program and assuring myself it was correct, I tended to just spew out code that was hopelessly broken, and gradually beat it into shape. Debugging, I was taught, was a kind of final pass where you caught typos and oversights. The way I worked, it seemed like programming consisted of debugging.

For a long time I felt bad about this, just as I once felt bad that I didn't hold my pencil the way they taught me to in elementary school. If I had only looked over at the other makers, the painters or the architects, I would have realized that there was a name for what I was doing: sketching. As far as I can tell, the way they taught me to program in college was all wrong. You should figure out programs as you're writing them, just as writers and painters and architects do.

Realizing this has real implications for software design. It means that a programming language should, above all, be malleable. A programming language is for [thinking](#) of programs, not for expressing programs you've already thought of. It should be a pencil, not a pen. Static typing would be a fine idea if people actually did write programs the way they taught me to in college. But that's not how any of the hackers I know write programs. We need a language that lets us scribble and smudge and smear, not a language where you have to sit with a teacup of types balanced on your knee and make polite conversation with a strict old aunt of a compiler.

While we're on the subject of static typing, identifying with the makers will save us from another problem that afflicts the sciences: math envy. Everyone in the sciences secretly believes that mathematicians are smarter than they are. I think mathematicians also believe this. At any rate, the result is that scientists tend to make their work look as mathematical as possible. In a field like physics this probably doesn't do much harm, but the further you get from the natural sciences, the more of a problem it becomes.

A page of formulas just looks so impressive. (Tip: for extra impressiveness, use Greek variables.) And so there is a great temptation to work on problems you can treat formally, rather than problems that are, say, important.

If hackers identified with other makers, like writers and painters, they wouldn't feel tempted to do this. Writers and painters don't suffer from math envy. They feel as if they're doing something completely unrelated. So are hackers, I think.

If universities and research labs keep hackers from doing the kind of work they

want to do, perhaps the place for them is in companies. Unfortunately, most companies won't let hackers do what they want either. Universities and research labs force hackers to be scientists, and companies force them to be engineers.

I only discovered this myself quite recently. When Yahoo bought Viaweb, they asked me what I wanted to do. I had never liked the business side very much, and said that I just wanted to hack. When I got to Yahoo, I found that what hacking meant to them was implementing software, not designing it. Programmers were seen as technicians who translated the visions (if that is the word) of product managers into code.

This seems to be the default plan in big companies. They do it because it decreases the standard deviation of the outcome. Only a small percentage of hackers can actually design software, and it's hard for the people running a company to pick these out. So instead of entrusting the future of the software to one brilliant hacker, most companies set things up so that it is designed by committee, and the hackers merely implement the design.

If you want to make money at some point, remember this, because this is one of the reasons startups win. Big companies want to decrease the standard deviation of design outcomes because they want to avoid disasters. But when you damp oscillations, you lose the high points as well as the low. This is not a problem for big companies, because they don't win by making great products. Big companies win by sucking less than other big companies.

So if you can figure out a way to get in a design war with a company big enough that its software is designed by product managers, they'll never be able to keep up with you. These opportunities are not easy to find, though. It's hard to engage a big company in a design war, just as it's hard to engage an opponent inside a castle in hand to hand combat. It would be pretty easy to write a better word processor than Microsoft Word, for example, but Microsoft, within the castle of their operating system monopoly, probably wouldn't even notice if you did.

The place to fight design wars is in new markets, where no one has yet managed to establish any fortifications. That's where you can win big by taking the bold approach to design, and having the same people both design and implement the product. Microsoft themselves did this at the start. So did Apple. And Hewlett-Packard. I suspect almost every successful startup has.

So one way to build great software is to start your own startup. There are two problems with this, though. One is that in a startup you have to do so much besides write software. At Viaweb I considered myself lucky if I got to hack a quarter of the time. And the things I had to do the other three quarters of the time ranged from tedious to terrifying. I have a benchmark for this, because I once had to leave a board meeting to have some cavities filled. I remember sitting back in the dentist's chair, waiting for the drill, and feeling like I was on vacation.

The other problem with startups is that there is not much overlap between the kind of software that makes money and the kind that's interesting to write. Programming languages are interesting to write, and Microsoft's first product was one, in fact, but no one will pay for programming languages now. If you want to make money, you tend to be forced to work on problems that are too nasty for anyone to solve for free.

All makers face this problem. Prices are determined by supply and demand, and there is just not as much demand for things that are fun to work on as there is for things that solve the mundane problems of individual customers. Acting in off-Broadway plays just doesn't pay as well as wearing a gorilla suit in someone's booth at a trade show. Writing novels doesn't pay as well as writing ad copy for garbage disposals. And hacking programming languages doesn't pay as well as figuring out how to connect some company's legacy database to their Web server.

I think the answer to this problem, in the case of software, is a concept known to nearly all makers: the day job. This phrase began with musicians, who perform at night. More generally, it means that you have one kind of work you do for money, and another for love.

Nearly all makers have day jobs early in their careers. Painters and writers notoriously do. If you're lucky you can get a day job that's closely related to your real work. Musicians often seem to work in record stores. A hacker working on some programming language or operating system might likewise be able to get a day job using it. [1]

When I say that the answer is for hackers to have day jobs, and work on beautiful software on the side, I'm not proposing this as a new idea. This is what open-source hacking is all about. What I'm saying is that open-source is probably the right model, because it has been independently confirmed by all the other makers.

It seems surprising to me that any employer would be reluctant to let hackers work on open-source projects. At Viaweb, we would have been reluctant to hire anyone who didn't. When we interviewed programmers, the main thing we cared about was what kind of software they wrote in their spare time. You can't do anything really well unless you love it, and if you love to hack you'll inevitably be working on projects of your own. [2]

Because hackers are makers rather than scientists, the right place to look for metaphors is not in the sciences, but among other kinds of makers. What else can painting teach us about hacking?

One thing we can learn, or at least confirm, from the example of painting is how to

learn to hack. You learn to paint mostly by doing it. Ditto for hacking. Most hackers don't learn to hack by taking college courses in programming. They learn to hack by writing programs of their own at age thirteen. Even in college classes, you learn to hack mostly by hacking. [3]

Because painters leave a trail of work behind them, you can watch them learn by doing. If you look at the work of a painter in chronological order, you'll find that each painting builds on things that have been learned in previous ones. When there's something in a painting that works very well, you can usually find version 1 of it in a smaller form in some earlier painting.

I think most makers work this way. Writers and architects seem to as well. Maybe it would be good for hackers to act more like painters, and regularly start over from scratch, instead of continuing to work for years on one project, and trying to incorporate all their later ideas as revisions.

The fact that hackers learn to hack by doing it is another sign of how different hacking is from the sciences. Scientists don't learn science by doing it, but by doing labs and problem sets. Scientists start out doing work that's perfect, in the sense that they're just trying to reproduce work someone else has already done for them. Eventually, they get to the point where they can do original work. Whereas hackers, from the start, are doing original work; it's just very bad. So hackers start original, and get good, and scientists start good, and get original.

The other way makers learn is from examples. For a painter, a museum is a reference library of techniques. For hundreds of years it has been part of the traditional education of painters to copy the works of the great masters, because copying forces you to look closely at the way a painting is made.

Writers do this too. Benjamin Franklin learned to write by summarizing the points in the essays of Addison and Steele and then trying to reproduce them. Raymond Chandler did the same thing with detective stories.

Hackers, likewise, can learn to program by looking at good programs-- not just at what they do, but the source code too. One of the less publicized benefits of the open-source movement is that it has made it easier to learn to program. When I learned to program, we had to rely mostly on examples in books. The one big chunk of code available then was Unix, but even this was not open source. Most of the people who read the source read it in illicit photocopies of John Lions' book, which though written in 1977 was not allowed to be published until 1996.

Another example we can take from painting is the way that paintings are created by gradual refinement. Paintings usually begin with a sketch. Gradually the details get filled in. But it is not merely a process of filling in. Sometimes the original plans

turn out to be mistaken. Countless paintings, when you look at them in xrays, turn out to have limbs that have been moved or facial features that have been readjusted.

Here's a case where we can learn from painting. I think hacking should work this way too. It's unrealistic to expect that the specifications for a program will be perfect. You're better off if you admit this up front, and write programs in a way that allows specifications to change on the fly.

(The structure of large companies makes this hard for them to do, so here is another place where startups have an advantage.)

Everyone by now presumably knows about the danger of premature optimization. I think we should be just as worried about premature design-- deciding too early what a program should do.

The right tools can help us avoid this danger. A good programming language should, like oil paint, make it easy to change your mind. Dynamic typing is a win here because you don't have to commit to specific data representations up front. But the key to flexibility, I think, is to make the language very abstract. The easiest program to change is one that's very short.

This sounds like a paradox, but a great painting has to be better than it has to be. For example, when Leonardo painted the portrait of Ginevra de Benci in the National Gallery, he put a juniper bush behind her head. In it he carefully painted each individual leaf. Many painters might have thought, this is just something to put in the background to frame her head. No one will look that closely at it.

Not Leonardo. How hard he worked on part of a painting didn't depend at all on how closely he expected anyone to look at it. He was like Michael Jordan. Relentless.

Relentlessness wins because, in the aggregate, unseen details become visible. When people walk by the portrait of Ginevra de Benci, their attention is often immediately arrested by it, even before they look at the label and notice that it says Leonardo da Vinci. All those unseen details combine to produce something that's just stunning, like a thousand barely audible voices all singing in tune.

Great software, likewise, requires a fanatical devotion to beauty. If you look inside good software, you find that parts no one is ever supposed to see are beautiful too. I'm not claiming I write great software, but I know that when it comes to code I behave in a way that would make me eligible for prescription drugs if I approached everyday life the same way. It drives me crazy to see code that's badly indented, or that uses ugly variable names.

If a hacker were a mere implementor, turning a spec into code, then he could just work his way through it from one end to the other like someone digging a ditch. But if the hacker is a creator, we have to take inspiration into account.

In hacking, like painting, work comes in cycles. Sometimes you get excited about some new project and you want to work sixteen hours a day on it. Other times nothing seems interesting.

To do good work you have to take these cycles into account, because they're affected by how you react to them. When you're driving a car with a manual transmission on a hill, you have to back off the clutch sometimes to avoid stalling. Backing off can likewise prevent ambition from stalling. In both painting and hacking there are some tasks that are terrifyingly ambitious, and others that are comfortingly routine. It's a good idea to save some easy tasks for moments when you would otherwise stall.

In hacking, this can literally mean saving up bugs. I like debugging: it's the one time that hacking is as straightforward as people think it is. You have a totally constrained problem, and all you have to do is solve it. Your program is supposed to do x. Instead it does y. Where does it go wrong? You know you're going to win in the end. It's as relaxing as painting a wall.

The example of painting can teach us not only how to manage our own work, but how to work together. A lot of the great art of the past is the work of multiple hands, though there may only be one name on the wall next to it in the museum. Leonardo was an apprentice in the workshop of Verrocchio and painted one of the angels in his [Baptism of Christ](). This sort of thing was the rule, not the exception. Michelangelo was considered especially dedicated for insisting on painting all the figures on the ceiling of the Sistine Chapel himself.

As far as I know, when painters worked together on a painting, they never worked on the same parts. It was common for the master to paint the principal figures and for assistants to paint the others and the background. But you never had one guy painting over the work of another.

I think this is the right model for collaboration in software too. Don't push it too far. When a piece of code is being hacked by three or four different people, no one of whom really owns it, it will end up being like a common-room. It will tend to feel bleak and abandoned, and accumulate cruft. The right way to collaborate, I think, is to divide projects into sharply defined modules, each with a definite owner, and with interfaces between them that are as carefully designed and, if possible, as articulated as programming languages.

Like painting, most software is intended for a human audience. And so hackers, like painters, must have empathy to do really great work. You have to be able to see things from the user's point of view.

When I was a kid I was always being told to look at things from someone else's point of view. What this always meant in practice was to do what someone else wanted, instead of what I wanted. This of course gave empathy a bad name, and I made a point of not cultivating it.

Boy, was I wrong. It turns out that looking at things from other people's point of view is practically the secret of success. It doesn't necessarily mean being self-sacrificing. Far from it. Understanding how someone else sees things doesn't imply that you'll act in his interest; in some situations-- in war, for example-- you want to do exactly the opposite. [4]

Most makers make things for a human audience. And to engage an audience you have to understand what they need. Nearly all the greatest paintings are paintings of people, for example, because people are what people are interested in.

Empathy is probably the single most important difference between a good hacker and a great one. Some hackers are quite smart, but when it comes to empathy are practically solipsists. It's hard for such people to design great software [5], because they can't see things from the user's point of view.

One way to tell how good people are at empathy is to watch them explain a technical question to someone without a technical background. We probably all know people who, though otherwise smart, are just comically bad at this. If someone asks them at a dinner party what a programming language is, they'll say something like ``Oh, a high-level language is what the compiler uses as input to generate object code.'' High-level language? Compiler? Object code? Someone who doesn't know what a programming language is obviously doesn't know what these things are, either.

Part of what software has to do is explain itself. So to write good software you have to understand how little users understand. They're going to walk up to the software with no preparation, and it had better do what they guess it will, because they're not going to read the manual. The best system I've ever seen in this respect was the original Macintosh, in 1985. It did what software almost never does: it just worked. [6]

Source code, too, should explain itself. If I could get people to remember just one quote about programming, it would be the one at the beginning of *Structure and Interpretation of Computer Programs.*

> Programs should be written for people to read, and only incidentally for machines to execute.

You need to have empathy not just for your users, but for your readers. It's in your

interest, because you'll be one of them. Many a hacker has written a program only to find on returning to it six months later that he has no idea how it works. I know several people who've sworn off Perl after such experiences. [7]

Lack of empathy is associated with intelligence, to the point that there is even something of a fashion for it in some places. But I don't think there's any correlation. You can do well in math and the natural sciences without having to learn empathy, and people in these fields tend to be smart, so the two qualities have come to be associated. But there are plenty of dumb people who are bad at empathy too. Just listen to the people who call in with questions on talk shows. They ask whatever it is they're asking in such a roundabout way that the hosts often have to rephrase the question for them.

So, if hacking works like painting and writing, is it as cool? After all, you only get one life. You might as well spend it working on something great.

Unfortunately, the question is hard to answer. There is always a big time lag in prestige. It's like light from a distant star. Painting has prestige now because of great work people did five hundred years ago. At the time, no one thought these paintings were as important as we do today. It would have seemed very odd to people at the time that Federico da Montefeltro, the Duke of Urbino, would one day be known mostly as the guy with the strange nose in a painting by Piero della Francesca.

So while I admit that hacking doesn't seem as cool as painting now, we should remember that painting itself didn't seem as cool in its glory days as it does now.

What we can say with some confidence is that these are the glory days of hacking. In most fields the great work is done early on. The paintings made between 1430 and 1500 are still unsurpassed. Shakespeare appeared just as professional theater was being born, and pushed the medium so far that every playwright since has had to live in his shadow. Albrecht Durer did the same thing with engraving, and Jane Austen with the novel.

Over and over we see the same pattern. A new medium appears, and people are so excited about it that they explore most of its possibilities in the first couple generations. Hacking seems to be in this phase now.

Painting was not, in Leonardo's time, as cool as his work helped make it. How cool hacking turns out to be will depend on what we can do with this new medium.

Notes

[1] The greatest damage that photography has done to painting may be the fact that it killed the best day job. Most of the great painters in history supported themselves by painting portraits.

[2] I've been told that Microsoft discourages employees from contributing to open-source projects, even in their spare time. But so many of the best hackers work on open-source projects now that the main effect of this policy may be to ensure that they won't be able to hire any first-rate programmers.

[3] What you learn about programming in college is much like what you learn about books or clothes or dating: what bad taste you had in high school.

[4] Here's an example of applied empathy. At Viaweb, if we couldn't decide between two alternatives, we'd ask, what would our competitors hate most? At one point a competitor added a feature to their software that was basically useless, but since it was one of few they had that we didn't, they made much of it in the trade press. We could have tried to explain that the feature was useless, but we decided it would annoy our competitor more if we just implemented it ourselves, so we hacked together our own version that afternoon.

[5] Except text editors and compilers. Hackers don't need empathy to design these, because they are themselves typical users.

[6] Well, almost. They overshot the available RAM somewhat, causing much inconvenient disk swapping, but this could be fixed within a few months by buying an additional disk drive.

[7] The way to make programs easy to read is not to stuff them with comments. I would take Abelson and Sussman's quote a step further. Programming languages should be designed to express algorithms, and only incidentally to tell computers how to execute them. A good programming language ought to be better for explaining software than English. You should only need comments when there is some kind of kludge you need to warn readers about, just as on a road there are only arrows on parts with unexpectedly sharp curves.

Thanks to Trevor Blackwell, Robert Morris, Dan Giffin, and Lisa Randall for reading drafts of this, and to Henry Leitner and Larry Finkelstein for inviting me to speak.

■

Japanese Translation

■

[Spanish Translation](#)

■

[German Translation](#)

■

[Portuguese Translation](#)

■

[Czech Translation](#)

■

[Why Good Design Comes from Bad Design](#)

■

[Knuth: Computer Programming as an Art](#)

You'll find this essay and 14 others in ***Hackers & Painters***.

Filters that Fight Back

August 2003

We may be able to improve the accuracy of Bayesian spam filters by having them follow links to see what's waiting at the other end. Richard Jowsey of death2spam now does this in borderline cases, and reports that it works well.

Why only do it in borderline cases? And why only do it once?

As I mentioned in Will Filters Kill Spam?, following all the urls in a spam would have an amusing side-effect. If popular email clients did this in order to filter spam, the spammer's servers would take a serious pounding. The more I think about this, the better an idea it seems. This isn't just amusing; it would be hard to imagine a more perfectly targeted counterattack on spammers.

So I'd like to suggest an additional feature to those working on spam filters: a "punish" mode which, if turned on, would spider every url in a suspected spam n times, where n could be set by the user. [1]

As many people have noted, one of the problems with the current email system is that it's too passive. It does whatever you tell it. So far all the suggestions for fixing the problem seem to involve new protocols. This one wouldn't.

If widely used, auto-retrieving spam filters would make the email system *rebound.* The huge volume of the spam, which has so far worked in the spammer's favor, would now work against him, like a branch snapping back in his face. Auto-retrieving spam filters would drive the spammer's costs up, and his sales down: his bandwidth usage would go through the roof, and his servers would grind to a halt under the load, which would make them unavailable to the people who would have responded to the spam.

Pump out a million emails an hour, get a million hits an hour on your servers.

We would want to ensure that this is only done to suspected spams. As a rule, any url sent to millions of people is likely to be a spam url, so submitting every http request in every email would work fine nearly all the time. But there are a few cases where this isn't true: the urls at the bottom of mails sent from free email services like Yahoo Mail and Hotmail, for example.

To protect such sites, and to prevent abuse, auto-retrieval should be combined with blacklists of spamvertised sites. Only sites on a blacklist would get crawled, and sites would be blacklisted only after being inspected by humans. The lifetime of a spam must be several hours at least, so it should be easy to update such a list in time to interfere with a spam promoting a new site. [2]

High-volume auto-retrieval would only be practical for users on high-bandwidth connections, but there are enough of those to cause spammers serious trouble. Indeed, this solution neatly mirrors the problem. The problem with spam is that in order to reach a few gullible people the spammer sends mail to everyone. The non-gullible recipients are merely collateral damage. But the non-gullible majority won't stop getting spam until they can stop (or threaten to stop) the gullible from responding to it. Auto-retrieving spam filters offer them a way to do this.

Would that kill spam? Not quite. The biggest spammers could probably protect their servers against auto-retrieving filters. However, the easiest and cheapest way for them to do it would be to include working unsubscribe links in their mails. And this would be a necessity for smaller fry, and for "legitimate" sites that hired spammers to promote them. So if auto-retrieving filters became widespread, they'd become auto-unsubscribing filters.

In this scenario, spam would, like OS crashes, viruses, and popups, become one of those plagues that only afflict people who don't bother to use the right software.

Notes

[1] Auto-retrieving filters will have to follow redirects, and should in some cases (e.g. a page that just says "click here") follow more than one level of links. Make sure too that the http requests are indistinguishable from those of popular Web browsers, including the order and referrer.

If the response doesn't come back within x amount of time, default to some fairly high spam probability.

Instead of making n constant, it might be a good idea to make it a function of the number of spams that have been seen mentioning the site. This would add a further level of protection against abuse and accidents.

[2] The original version of this article used the term "whitelist" instead of "blacklist". Though they were to work like blacklists, I preferred to call them whitelists because it might make them less vulnerable to legal attack. This just seems to have confused readers, though.

There should probably be multiple blacklists. A single point of failure would be vulnerable both to attack and abuse.

·

[FFB FAQ](#)

·

[Japanese Translation](#)

·

[A Perl FFB](#)

·

[Lycos DDoS@Home](#)

What You Can't Say

January 2004

Have you ever seen an old photo of yourself and been embarrassed at the way you looked? *Did we actually dress like that?* We did. And we had no idea how silly we looked. It's the nature of fashion to be invisible, in the same way the movement of the earth is invisible to all of us riding on it.

What scares me is that there are moral fashions too. They're just as arbitrary, and just as invisible to most people. But they're much more dangerous. Fashion is mistaken for good design; moral fashion is mistaken for good. Dressing oddly gets you laughed at. Violating moral fashions can get you fired, ostracized, imprisoned, or even killed.

If you could travel back in a time machine, one thing would be true no matter where you went: you'd have to watch what you said. Opinions we consider harmless could have gotten you in big trouble. I've already said at least one thing that would have gotten me in big trouble in most of Europe in the seventeenth century, and did get Galileo in big trouble when he said it � that the earth moves. [1]

It seems to be a constant throughout history: In every period, people believed things that were just ridiculous, and believed them so strongly that you would have gotten in terrible trouble for saying otherwise.

Is our time any different? To anyone who has read any amount of history, the answer is almost certainly no. It would be a remarkable coincidence if ours were the first era to get everything just right.

It's tantalizing to think we believe things that people in the future will find ridiculous. What *would* someone coming back to visit us in a time machine have to be careful not to say? That's what I want to study here. But I want to do more than just shock everyone with the heresy du jour. I want to find general recipes for discovering what you can't say, in any era.

The Conformist Test

Let's start with a test: Do you have any opinions that you would be reluctant to express in front of a group of your peers?

If the answer is no, you might want to stop and think about that. If everything you believe is something you're supposed to believe, could that possibly be a coincidence? Odds are it isn't. Odds are you just think what you're told.

The other alternative would be that you independently considered every question and came up with the exact same answers that are now considered acceptable. That seems unlikely, because you'd also have to make the same mistakes. Mapmakers deliberately put slight mistakes in their maps so they can tell when someone copies them. If another map has the same mistake, that's very convincing evidence.

Like every other era in history, our moral map almost certainly contains a few mistakes. And anyone who makes the same mistakes probably didn't do it by accident. It would be like someone claiming they had independently decided in 1972 that bell-bottom jeans were a good idea.

If you believe everything you're supposed to now, how can you be sure you wouldn't also have believed everything you were supposed to if you had grown up among the plantation owners of the pre-Civil War South, or in Germany in the 1930s � or among the Mongols in 1200, for that matter? Odds are you would have.

Back in the era of terms like "well-adjusted," the idea seemed to be that there was something wrong with you if you thought things you didn't dare say out loud. This seems backward. Almost certainly, there is something wrong with you if you *don't* think things you don't dare say out loud.

Trouble

What can't we say? One way to find these ideas is simply to look at things people do say, and get in trouble for. [2]

Of course, we're not just looking for things we can't say. We're looking for things we can't say that are true, or at least have enough chance of being true that the question should remain open. But many of the things people get in trouble for saying probably do make it over this second, lower threshold. No one gets in trouble for saying that 2 + 2 is 5, or that people in Pittsburgh are ten feet tall. Such obviously false statements might be treated as jokes, or at worst as evidence of insanity, but they are not likely to make anyone mad. The statements that make people mad are the ones they worry might be believed. I suspect the statements that make people maddest are those they worry might be true.

If Galileo had said that people in Padua were ten feet tall, he would have been regarded as a harmless eccentric. Saying the earth orbited the sun was another

matter. The church knew this would set people thinking.

Certainly, as we look back on the past, this rule of thumb works well. A lot of the statements people got in trouble for seem harmless now. So it's likely that visitors from the future would agree with at least some of the statements that get people in trouble today. Do we have no Galileos? Not likely.

To find them, keep track of opinions that get people in trouble, and start asking, could this be true? Ok, it may be heretical (or whatever modern equivalent), but might it also be true?

Heresy

This won't get us all the answers, though. What if no one happens to have gotten in trouble for a particular idea yet? What if some idea would be so radioactively controversial that no one would dare express it in public? How can we find these too?

Another approach is to follow that word, heresy. In every period of history, there seem to have been labels that got applied to statements to shoot them down before anyone had a chance to ask if they were true or not. "Blasphemy", "sacrilege", and "heresy" were such labels for a good part of western history, as in more recent times "indecent", "improper", and "unamerican" have been. By now these labels have lost their sting. They always do. By now they're mostly used ironically. But in their time, they had real force.

The word "defeatist", for example, has no particular political connotations now. But in Germany in 1917 it was a weapon, used by Ludendorff in a purge of those who favored a negotiated peace. At the start of World War II it was used extensively by Churchill and his supporters to silence their opponents. In 1940, any argument against Churchill's aggressive policy was "defeatist". Was it right or wrong? Ideally, no one got far enough to ask that.

We have such labels today, of course, quite a lot of them, from the all-purpose "inappropriate" to the dreaded "divisive." In any period, it should be easy to figure out what such labels are, simply by looking at what people call ideas they disagree with besides untrue. When a politician says his opponent is mistaken, that's a straightforward criticism, but when he attacks a statement as "divisive" or "racially insensitive" instead of arguing that it's false, we should start paying attention.

So another way to figure out which of our taboos future generations will laugh at is to start with the labels. Take a label � "sexist", for example � and try to think of some ideas that would be called that. Then for each ask, might this be true?

Just start listing ideas at random? Yes, because they won't really be random. The ideas that come to mind first will be the most plausible ones. They'll be things you've already noticed but didn't let yourself think.

In 1989 some clever researchers tracked the eye movements of radiologists as they scanned chest images for signs of lung cancer. [3] They found that even when the radiologists missed a cancerous lesion, their eyes had usually paused at the site of it. Part of their brain knew there was something there; it just didn't percolate all the way up into conscious knowledge. I think many interesting heretical thoughts are already mostly formed in our minds. If we turn off our self-censorship temporarily, those will be the first to emerge.

Time and Space

If we could look into the future it would be obvious which of our taboos they'd laugh at. We can't do that, but we can do something almost as good: we can look into the past. Another way to figure out what we're getting wrong is to look at what used to be acceptable and is now unthinkable.

Changes between the past and the present sometimes do represent progress. In a field like physics, if we disagree with past generations it's because we're right and they're wrong. But this becomes rapidly less true as you move away from the certainty of the hard sciences. By the time you get to social questions, many changes are just fashion. The age of consent fluctuates like hemlines.

We may imagine that we are a great deal smarter and more virtuous than past generations, but the more history you read, the less likely this seems. People in past times were much like us. Not heroes, not barbarians. Whatever their ideas were, they were ideas reasonable people could believe.

So here is another source of interesting heresies. Diff present ideas against those of various past cultures, and see what you get. [4] Some will be shocking by present standards. Ok, fine; but which might also be true?

You don't have to look into the past to find big differences. In our own time, different societies have wildly varying ideas of what's ok and what isn't. So you can try diffing other cultures' ideas against ours as well. (The best way to do that is to visit them.) Any idea that's considered harmless in a significant percentage of times and places, and yet is taboo in ours, is a candidate for something we're mistaken about.

For example, at the high water mark of political correctness in the early 1990s, Harvard distributed to its faculty and staff a brochure saying, among other things, that it was inappropriate to compliment a colleague or student's clothes. No more "nice shirt." I think this principle is rare among the world's cultures, past or present. There are probably more where it's considered especially polite to compliment someone's clothing than where it's considered improper. Odds are this is, in a mild form, an example of one of the taboos a visitor from the future would have to be careful to avoid if he happened to set his time machine for Cambridge, Massachusetts, 1992. [5]

Prigs

Of course, if they have time machines in the future they'll probably have a separate reference manual just for Cambridge. This has always been a fussy place, a town of i dotters and t crossers, where you're liable to get both your grammar and your ideas corrected in the same conversation. And that suggests another way to find taboos. Look for prigs, and see what's inside their heads.

Kids' heads are repositories of all our taboos. It seems fitting to us that kids' ideas should be bright and clean. The picture we give them of the world is not merely simplified, to suit their developing minds, but sanitized as well, to suit our ideas of what kids ought to think. [6]

You can see this on a small scale in the matter of dirty words. A lot of my friends are starting to have children now, and they're all trying not to use words like "fuck" and "shit" within baby's hearing, lest baby start using these words too. But these words are part of the language, and adults use them all the time. So parents are giving their kids an inaccurate idea of the language by not using them. Why do they do this? Because they don't think it's fitting that kids should use the whole language. We like children to seem innocent. [7]

Most adults, likewise, deliberately give kids a misleading view of the world. One of the most obvious examples is Santa Claus. We think it's cute for little kids to believe in Santa Claus. I myself think it's cute for little kids to believe in Santa Claus. But one wonders, do we tell them this stuff for their sake, or for ours?

I'm not arguing for or against this idea here. It is probably inevitable that parents should want to dress up their kids' minds in cute little baby outfits. I'll probably do it myself. The important thing for our purposes is that, as a result, a well brought-up teenage kid's brain is a more or less complete collection of all our taboos � and in mint condition, because they're untainted by experience. Whatever we think that will later turn out to be ridiculous, it's almost certainly inside that head.

How do we get at these ideas? By the following thought experiment. Imagine a kind of latter-day Conrad character who has worked for a time as a mercenary in Africa, for a time as a doctor in Nepal, for a time as the manager of a nightclub in Miami. The specifics don't matter � just someone who has seen a lot. Now imagine comparing what's inside this guy's head with what's inside the head of a well-behaved sixteen year old girl from the suburbs. What does he think that would shock her? He knows the world; she knows, or at least embodies, present taboos. Subtract one from the other, and the result is what we can't say.

Mechanism

I can think of one more way to figure out what we can't say: to look at how taboos are created. How do moral fashions arise, and why are they adopted? If we can understand this mechanism, we may be able to see it at work in our own time.

Moral fashions don't seem to be created the way ordinary fashions are. Ordinary

fashions seem to arise by accident when everyone imitates the whim of some influential person. The fashion for broad-toed shoes in late fifteenth century Europe began because Charles VIII of France had six toes on one foot. The fashion for the name Gary began when the actor Frank Cooper adopted the name of a tough mill town in Indiana. Moral fashions more often seem to be created deliberately. When there's something we can't say, it's often because some group doesn't want us to.

The prohibition will be strongest when the group is nervous. The irony of Galileo's situation was that he got in trouble for repeating Copernicus's ideas. Copernicus himself didn't. In fact, Copernicus was a canon of a cathedral, and dedicated his book to the pope. But by Galileo's time the church was in the throes of the Counter-Reformation and was much more worried about unorthodox ideas.

To launch a taboo, a group has to be poised halfway between weakness and power. A confident group doesn't need taboos to protect it. It's not considered improper to make disparaging remarks about Americans, or the English. And yet a group has to be powerful enough to enforce a taboo. Coprophiles, as of this writing, don't seem to be numerous or energetic enough to have had their interests promoted to a lifestyle.

I suspect the biggest source of moral taboos will turn out to be power struggles in which one side only barely has the upper hand. That's where you'll find a group powerful enough to enforce taboos, but weak enough to need them.

Most struggles, whatever they're really about, will be cast as struggles between competing ideas. The English Reformation was at bottom a struggle for wealth and power, but it ended up being cast as a struggle to preserve the souls of Englishmen from the corrupting influence of Rome. It's easier to get people to fight for an idea. And whichever side wins, their ideas will also be considered to have triumphed, as if God wanted to signal his agreement by selecting that side as the victor.

We often like to think of World War II as a triumph of freedom over totalitarianism. We conveniently forget that the Soviet Union was also one of the winners.

I'm not saying that struggles are never about ideas, just that they will always be made to seem to be about ideas, whether they are or not. And just as there is nothing so unfashionable as the last, discarded fashion, there is nothing so wrong as the principles of the most recently defeated opponent. Representational art is only now recovering from the approval of both Hitler and Stalin. [8]

Although moral fashions tend to arise from different sources than fashions in clothing, the mechanism of their adoption seems much the same. The early adopters will be driven by ambition: self-consciously cool people who want to distinguish themselves from the common herd. As the fashion becomes established they'll be joined by a second, much larger group, driven by fear. [9] This second group adopt the fashion not because they want to stand out but because they are afraid of standing out.

So if you want to figure out what we can't say, look at the machinery of fashion and try to predict what it would make unsayable. What groups are powerful but nervous, and what ideas would they like to suppress? What ideas were tarnished by association when they ended up on the losing side of a recent struggle? If a self-consciously cool person wanted to differentiate himself from preceding fashions (e.g. from his parents), which of their ideas would he tend to reject? What are conventional-minded people afraid of saying?

This technique won't find us all the things we can't say. I can think of some that aren't the result of any recent struggle. Many of our taboos are rooted deep in the past. But this approach, combined with the preceding four, will turn up a good number of unthinkable ideas.

Why

Some would ask, why would one want to do this? Why deliberately go poking around among nasty, disreputable ideas? Why look under rocks?

I do it, first of all, for the same reason I did look under rocks as a kid: plain curiosity. And I'm especially curious about anything that's forbidden. Let me see and decide for myself.

Second, I do it because I don't like the idea of being mistaken. If, like other eras, we believe things that will later seem ridiculous, I want to know what they are so that I, at least, can avoid believing them.

Third, I do it because it's good for the brain. To do good work you need a brain that can go anywhere. And you especially need a brain that's in the habit of going where it's not supposed to.

Great work tends to grow out of ideas that others have overlooked, and no idea is so overlooked as one that's unthinkable. Natural selection, for example. It's so simple. Why didn't anyone think of it before? Well, that is all too obvious. Darwin himself was careful to tiptoe around the implications of his theory. He wanted to spend his time thinking about biology, not arguing with people who accused him of being an atheist.

In the sciences, especially, it's a great advantage to be able to question assumptions. The m.o. of scientists, or at least of the good ones, is precisely that: look for places where conventional wisdom is broken, and then try to pry apart the cracks and see what's underneath. That's where new theories come from.

A good scientist, in other words, does not merely ignore conventional wisdom, but makes a special effort to break it. Scientists go looking for trouble. This should be the m.o. of any scholar, but scientists seem much more willing to look under rocks. [10]

Why? It could be that the scientists are simply smarter; most physicists could, if necessary, make it through a PhD program in French literature, but few professors of French literature could make it through a PhD program in physics. Or it could be because it's clearer in the sciences whether theories are true or false, and this makes scientists bolder. (Or it could be that, because it's clearer in the sciences whether theories are true or false, you have to be smart to get jobs as a scientist, rather than just a good politician.)

Whatever the reason, there seems a clear correlation between intelligence and willingness to consider shocking ideas. This isn't just because smart people actively work to find holes in conventional thinking. I think conventions also have less hold over them to start with. You can see that in the way they dress.

It's not only in the sciences that heresy pays off. In any competitive field, you can win big by seeing things that others daren't. And in every field there are probably heresies few dare utter. Within the US car industry there is a lot of hand-wringing now about declining market share. Yet the cause is so obvious that any observant outsider could explain it in a second: they make bad cars. And they have for so long that by now the US car brands are antibrands � something you'd buy a car despite, not because of. Cadillac stopped being the Cadillac of cars in about 1970. And yet I suspect no one dares say this. [11] Otherwise these companies would have tried to fix the problem.

Training yourself to think unthinkable thoughts has advantages beyond the thoughts themselves. It's like stretching. When you stretch before running, you put your body into positions much more extreme than any it will assume during the run. If you can think things so outside the box that they'd make people's hair stand on end, you'll have no trouble with the small trips outside the box that people call innovative.

Pensieri Stretti

When you find something you can't say, what do you do with it? My advice is, don't say it. Or at least, pick your battles.

Suppose in the future there is a movement to ban the color yellow. Proposals to paint anything yellow are denounced as "yellowist", as is anyone suspected of liking the color. People who like orange are tolerated but viewed with suspicion. Suppose you realize there is nothing wrong with yellow. If you go around saying this, you'll be denounced as a yellowist too, and you'll find yourself having a lot of arguments with anti-yellowists. If your aim in life is to rehabilitate the color yellow, that may be what you want. But if you're mostly interested in other questions, being labelled as a yellowist will just be a distraction. Argue with idiots, and you become an idiot.

The most important thing is to be able to think what you want, not to say what you want. And if you feel you have to say everything you think, it may inhibit you from thinking improper thoughts. I think it's better to follow the opposite policy. Draw a

sharp line between your thoughts and your speech. Inside your head, anything is allowed. Within my head I make a point of encouraging the most outrageous thoughts I can imagine. But, as in a secret society, nothing that happens within the building should be told to outsiders. The first rule of Fight Club is, you do not talk about Fight Club.

When Milton was going to visit Italy in the 1630s, Sir Henry Wootton, who had been ambassador to Venice, told him his motto should be *"i pensieri stretti & il viso sciolto."* Closed thoughts and an open face. Smile at everyone, and don't tell them what you're thinking. This was wise advice. Milton was an argumentative fellow, and the Inquisition was a bit restive at that time. But I think the difference between Milton's situation and ours is only a matter of degree. Every era has its heresies, and if you don't get imprisoned for them you will at least get in enough trouble that it becomes a complete distraction.

I admit it seems cowardly to keep quiet. When I read about the harassment to which the Scientologists subject their critics [12], or that pro-Israel groups are "compiling dossiers" on those who speak out against Israeli human rights abuses [13], or about people being sued for violating the DMCA [14], part of me wants to say, "All right, you bastards, bring it on." The problem is, there are so many things you can't say. If you said them all you'd have no time left for your real work. You'd have to turn into Noam Chomsky. [15]

The trouble with keeping your thoughts secret, though, is that you lose the advantages of discussion. Talking about an idea leads to more ideas. So the optimal plan, if you can manage it, is to have a few trusted friends you can speak openly to. This is not just a way to develop ideas; it's also a good rule of thumb for choosing friends. The people you can say heretical things to without getting jumped on are also the most interesting to know.

Viso Sciolto?

I don't think we need the *viso sciolto* so much as the *pensieri stretti.* Perhaps the best policy is to make it plain that you don't agree with whatever zealotry is current in your time, but not to be too specific about what you disagree with. Zealots will try to draw you out, but you don't have to answer them. If they try to force you to treat a question on their terms by asking "are you with us or against us?" you can always just answer "neither".

Better still, answer "I haven't decided." That's what Larry Summers did when a group tried to put him in this position. Explaining himself later, he said "I don't do litmus tests." [16] A lot of the questions people get hot about are actually quite complicated. There is no prize for getting the answer quickly.

If the anti-yellowists seem to be getting out of hand and you want to fight back, there are ways to do it without getting yourself accused of being a yellowist. Like skirmishers in an ancient army, you want to avoid directly engaging the main body of the enemy's troops. Better to harass them with arrows from a distance.

One way to do this is to ratchet the debate up one level of abstraction. If you argue against censorship in general, you can avoid being accused of whatever heresy is contained in the book or film that someone is trying to censor. You can attack labels with meta-labels: labels that refer to the use of labels to prevent discussion. The spread of the term "political correctness" meant the beginning of the end of political correctness, because it enabled one to attack the phenomenon as a whole without being accused of any of the specific heresies it sought to suppress.

Another way to counterattack is with metaphor. Arthur Miller undermined the House Un-American Activities Committee by writing a play, "The Crucible," about the Salem witch trials. He never referred directly to the committee and so gave them no way to reply. What could HUAC do, defend the Salem witch trials? And yet Miller's metaphor stuck so well that to this day the activities of the committee are often described as a "witch-hunt."

Best of all, probably, is humor. Zealots, whatever their cause, invariably lack a sense of humor. They can't reply in kind to jokes. They're as unhappy on the territory of humor as a mounted knight on a skating rink. Victorian prudishness, for example, seems to have been defeated mainly by treating it as a joke. Likewise its reincarnation as political correctness. "I am glad that I managed to write 'The Crucible,'" Arthur Miller wrote, "but looking back I have often wished I'd had the temperament to do an absurd comedy, which is what the situation deserved." [17]

ABQ

A Dutch friend says I should use Holland as an example of a tolerant society. It's true they have a long tradition of comparative open-mindedness. For centuries the low countries were the place to go to say things you couldn't say anywhere else, and this helped to make the region a center of scholarship and industry (which have been closely tied for longer than most people realize). Descartes, though claimed by the French, did much of his thinking in Holland.

And yet, I wonder. The Dutch seem to live their lives up to their necks in rules and regulations. There's so much you can't do there; is there really nothing you can't say?

Certainly the fact that they value open-mindedness is no guarantee. Who thinks they're not open-minded? Our hypothetical prim miss from the suburbs thinks she's open-minded. Hasn't she been taught to be? Ask anyone, and they'll say the same thing: they're pretty open-minded, though they draw the line at things that are really wrong. (Some tribes may avoid "wrong" as judgemental, and may instead use a more neutral sounding euphemism like "negative" or "destructive".)

When people are bad at math, they know it, because they get the wrong answers on tests. But when people are bad at open-mindedness they don't know it. In fact they tend to think the opposite. Remember, it's the nature of fashion to be

invisible. It wouldn't work otherwise. Fashion doesn't seem like fashion to someone in the grip of it. It just seems like the right thing to do. It's only by looking from a distance that we see oscillations in people's idea of the right thing to do, and can identify them as fashions.

Time gives us such distance for free. Indeed, the arrival of new fashions makes old fashions easy to see, because they seem so ridiculous by contrast. From one end of a pendulum's swing, the other end seems especially far away.

To see fashion in your own time, though, requires a conscious effort. Without time to give you distance, you have to create distance yourself. Instead of being part of the mob, stand as far away from it as you can and watch what it's doing. And pay especially close attention whenever an idea is being suppressed. Web filters for children and employees often ban sites containing pornography, violence, and hate speech. What counts as pornography and violence? And what, exactly, is "hate speech?" This sounds like a phrase out of *1984.*

Labels like that are probably the biggest external clue. If a statement is false, that's the worst thing you can say about it. You don't need to say that it's heretical. And if it isn't false, it shouldn't be suppressed. So when you see statements being attacked as x-ist or y-ic (substitute your current values of x and y), whether in 1630 or 2030, that's a sure sign that something is wrong. When you hear such labels being used, ask why.

Especially if you hear yourself using them. It's not just the mob you need to learn to watch from a distance. You need to be able to watch your own thoughts from a distance. That's not a radical idea, by the way; it's the main difference between children and adults. When a child gets angry because he's tired, he doesn't know what's happening. An adult can distance himself enough from the situation to say "never mind, I'm just tired." I don't see why one couldn't, by a similar process, learn to recognize and discount the effects of moral fashions.

You have to take that extra step if you want to think clearly. But it's harder, because now you're working against social customs instead of with them. Everyone encourages you to grow up to the point where you can discount your own bad moods. Few encourage you to continue to the point where you can discount society's bad moods.

How can you see the wave, when you're the water? Always be questioning. That's the only defence. What can't you say? And why?

Notes

heresy. Needless to say they bear no blame for opinions expressed in it, and especially for opinions *not* expressed in it.

■

[Re: What You Can't Say](#)

■

[Labels](#)

■

[Japanese Translation](#)

■

[French Translation](#)

■

[German Translation](#)

■

[Dutch Translation](#)

■

[Romanian Translation](#)

■

[Hebrew Translation](#)

■

■

[A Blacklist for "Excuse Makers"](#)

■

[What You Can't Say Will Hurt You](#)

The Word "Hacker"

April 2004

To the popular press, "hacker" means someone who breaks into computers. Among programmers it means a good programmer. But the two meanings are connected. To programmers, "hacker" connotes mastery in the most literal sense: someone who can make a computer do what he wants—whether the computer wants to or not.

To add to the confusion, the noun "hack" also has two senses. It can be either a compliment or an insult. It's called a hack when you do something in an ugly way. But when you do something so clever that you somehow beat the system, that's also called a hack. The word is used more often in the former than the latter sense, probably because ugly solutions are more common than brilliant ones.

Believe it or not, the two senses of "hack" are also connected. Ugly and imaginative solutions have something in common: they both break the rules. And there is a gradual continuum between rule breaking that's merely ugly (using duct tape to attach something to your bike) and rule breaking that is brilliantly imaginative (discarding Euclidean space).

Hacking predates computers. When he was working on the Manhattan Project, Richard Feynman used to amuse himself by breaking into safes containing secret documents. This tradition continues today. When we were in grad school, a hacker friend of mine who spent too much time around MIT had his own lock picking kit. (He now runs a hedge fund, a not unrelated enterprise.)

It is sometimes hard to explain to authorities why one would want to do such things. Another friend of mine once got in trouble with the government for breaking into computers. This had only recently been declared a crime, and the FBI found that their usual investigative technique didn't work. Police investigation apparently begins with a motive. The usual motives are few: drugs, money, sex,

revenge. Intellectual curiosity was not one of the motives on the FBI's list. Indeed, the whole concept seemed foreign to them.

Those in authority tend to be annoyed by hackers' general attitude of disobedience. But that disobedience is a byproduct of the qualities that make them good programmers. They may laugh at the CEO when he talks in generic corporate newspeech, but they also laugh at someone who tells them a certain problem can't be solved. Suppress one, and you suppress the other.

This attitude is sometimes affected. Sometimes young programmers notice the eccentricities of eminent hackers and decide to adopt some of their own in order to seem smarter. The fake version is not merely annoying; the prickly attitude of these posers can actually slow the process of innovation.

But even factoring in their annoying eccentricities, the disobedient attitude of hackers is a net win. I wish its advantages were better understood.

For example, I suspect people in Hollywood are simply mystified by hackers' attitudes toward copyrights. They are a perennial topic of heated discussion on Slashdot. But why should people who program computers be so concerned about copyrights, of all things?

Partly because some companies use *mechanisms* to prevent copying. Show any hacker a lock and his first thought is how to pick it. But there is a deeper reason that hackers are alarmed by measures like copyrights and patents. They see increasingly aggressive measures to protect "intellectual property" as a threat to the intellectual freedom they need to do their job. And they are right.

It is by poking about inside current technology that hackers get ideas for the next generation. No thanks, intellectual homeowners may say, we don't need any outside help. But they're wrong. The next generation of computer technology has often—perhaps more often than not—been developed by outsiders.

In 1977 there was no doubt some group within IBM developing what they expected to be the next generation of business computer. They were mistaken. The next generation of business computer was being developed on entirely different lines by two long-haired guys called Steve in a [garage](#) in Los Altos. At about the same time, the powers that be were cooperating to develop the official next generation operating system, Multics. But two guys who thought Multics excessively complex went off and wrote their own. They gave it a name that was a joking reference to Multics: Unix.

The latest intellectual property laws impose unprecedented restrictions on the sort of poking around that leads to new ideas. In the past, a competitor might use patents to prevent you from selling a copy of something they made, but they couldn't prevent you from taking one apart to see how it worked. The latest laws make this a crime. How are we to develop new technology if we can't study current technology to figure out how to improve it?

Ironically, hackers have brought this on themselves. Computers are responsible for the problem. The control systems inside machines used to be physical: gears and levers and cams. Increasingly, the brains (and thus the value) of products is in software. And by this I mean software in the general sense: i.e. data. A song on an LP is physically stamped into the plastic. A song on an iPod's disk is merely stored on it.

Data is by definition easy to copy. And the Internet makes copies easy to distribute. So it is no wonder companies are afraid. But, as so often happens, fear has clouded their judgement. The government has responded with draconian laws to protect intellectual property. They probably mean well. But they may not realize that such laws will do more harm than good.

Why are programmers so violently opposed to these laws? If I were a legislator, I'd be interested in this mystery—for the same reason that, if I were a farmer and suddenly heard a lot of squawking coming from my hen house one night, I'd want to go out and investigate. Hackers are not stupid, and unanimity is very rare in this world. So if they're all squawking, perhaps there is something amiss.

Could it be that such laws, though intended to protect America, will actually harm it? Think about it. There is something very *American* about Feynman breaking into safes during the Manhattan Project. It's hard to imagine the authorities having a sense of humor about such things over in Germany at that time. Maybe it's not a coincidence.

Hackers are unruly. That is the essence of hacking. And it is also the essence of Americanness. It is no accident that Silicon Valley is in America, and not France, or Germany, or England, or Japan. In those countries, people color inside the lines.

I lived for a while in Florence. But after I'd been there a few months I realized that what I'd been unconsciously hoping to find there was back in the place I'd just left. The reason Florence is famous is that in 1450, it was New York. In 1450 it was filled with the kind of turbulent and ambitious people you find now in America. (So I went back to America.)

It is greatly to America's advantage that it is a congenial atmosphere for the right sort of unruliness—that it is a home not just for the smart, but for smart-alecks. And hackers are invariably smart-alecks. If we had a national holiday, it would be April 1st. It says a great deal about our work that we use the same word for a brilliant or a horribly cheesy solution. When we cook one up we're not always 100% sure which kind it is. But as long as it has the right sort of wrongness, that's a promising sign. It's odd that people think of programming as precise and methodical. *Computers* are precise and methodical. Hacking is something you do with a gleeful laugh.

In our world some of the most characteristic solutions are not far removed from practical jokes. IBM was no doubt rather surprised by the consequences of the

licensing deal for DOS, just as the hypothetical "adversary" must be when Michael Rabin solves a problem by redefining it as one that's easier to solve.

Smart-alecks have to develop a keen sense of how much they can [get away] with. And lately hackers have sensed a change in the atmosphere. Lately hackerliness seems rather frowned upon.

To hackers the recent contraction in civil liberties seems especially ominous. That must also mystify outsiders. Why should we care especially about civil liberties? Why programmers, more than dentists or salesmen or landscapers?

Let me put the case in terms a government official would appreciate. Civil liberties are not just an ornament, or a quaint American tradition. Civil liberties make countries rich. If you made a graph of GNP per capita vs. civil liberties, you'd notice a definite trend. Could civil liberties really be a cause, rather than just an effect? I think so. I think a society in which people can do and say what they want will also tend to be one in which the most efficient solutions win, rather than those sponsored by the most influential people. Authoritarian countries become corrupt; corrupt countries become poor; and poor countries are weak. It seems to me there is a Laffer curve for government power, just as for tax revenues. At least, it seems likely enough that it would be stupid to try the experiment and find out. Unlike high tax rates, you can't repeal totalitarianism if it turns out to be a mistake.

This is why hackers worry. The government spying on people doesn't literally make programmers write worse code. It just leads eventually to a world in which bad ideas win. And because this is so important to hackers, they're especially sensitive to it. They can sense totalitarianism approaching from a distance, as animals can sense an approaching thunderstorm.

It would be ironic if, as hackers fear, recent measures intended to protect national security and intellectual property turned out to be a missile aimed right at what makes America successful. But it would not be the first time that measures taken in an atmosphere of panic had the opposite of the intended effect.

There is such a thing as Americanness. There's nothing like living abroad to teach you that. And if you want to know whether something will nurture or squash this quality, it would be hard to find a better focus group than hackers, because they come closest of any group I know to embodying it. Closer, probably, than the men running our government, who for all their talk of patriotism remind me more of Richelieu or Mazarin than Thomas Jefferson or George Washington.

When you read what the founding fathers had to say for themselves, they sound more like hackers. "The spirit of resistance to government," Jefferson wrote, "is so valuable on certain occasions, that I wish it always to be kept alive."

Imagine an American president saying that today. Like the remarks of an outspoken old grandmother, the sayings of the founding fathers have embarrassed generations of their less confident successors. They remind us where we come

from. They remind us that it is the people who break rules that are the source of America's wealth and power.

Those in a position to impose rules naturally want them to be obeyed. But be careful what you ask for. You might get it.

(The image shows Steves Jobs and Wozniak with a "blue box." Photo by Margret Wozniak. Reproduced by permission of Steve Wozniak.)

■

Portuguese Translation

■

Hebrew Translation

■

Romanian Translation

You'll find this essay and 14 others in *Hackers & Painters*.

How to Make Wealth

May 2004

(This essay was originally published in [Hackers & Painters](#).)

If you wanted to get rich, how would you do it? I think your best bet would be to start or join a startup. That's been a reliable way to get rich for hundreds of years. The word "startup" dates from the 1960s, but what happens in one is very similar to the venture-backed trading voyages of the Middle Ages.

Startups usually involve technology, so much so that the phrase "high-tech startup" is almost redundant. A startup is a small company that takes on a hard technical problem.

Lots of people get rich knowing nothing more than that. You don't have to know physics to be a good pitcher. But I think it could give you an edge to understand the underlying principles. Why do startups have to be small? Will a startup inevitably stop being a startup as it grows larger? And why do they so often work on developing new technology? Why are there so many startups selling new drugs or computer software, and none selling corn oil or laundry detergent?

The Proposition

Economically, you can think of a startup as a way to compress your whole working life into a few years. Instead of working at a low intensity for forty years, you work as hard as you possibly can for four. This pays especially well in technology, where you earn a premium for working fast.

Here is a brief sketch of the economic proposition. If you're a good hacker in your mid twenties, you can get a job paying about $80,000 per year. So on average such a hacker must be able to do at least $80,000 worth of work per year for the company just to break even. You could probably work twice as many hours as a corporate employee, and if you focus you can probably get three times as much done in an hour. [1] You should get another multiple of two, at least, by eliminating the drag of the pointy-haired middle manager who would be your boss in a big company. Then there is one more multiple: how much smarter are you than your job description expects you to be? Suppose another multiple of three. Combine all these multipliers, and I'm claiming you could be 36 times more

productive than you're expected to be in a random corporate job. [2] If a fairly good hacker is worth $80,000 a year at a big company, then a smart hacker working very hard without any corporate bullshit to slow him down should be able to do work worth about $3 million a year.

Like all back-of-the-envelope calculations, this one has a lot of wiggle room. I wouldn't try to defend the actual numbers. But I stand by the structure of the calculation. I'm not claiming the multiplier is precisely 36, but it is certainly more than 10, and probably rarely as high as 100.

If $3 million a year seems high, remember that we're talking about the limit case: the case where you not only have zero leisure time but indeed work so hard that you endanger your health.

Startups are not magic. They don't change the laws of wealth creation. They just represent a point at the far end of the curve. There is a conservation law at work here: if you want to make a million dollars, you have to endure a million dollars' worth of pain. For example, one way to make a million dollars would be to work for the Post Office your whole life, and save every penny of your salary. Imagine the stress of working for the Post Office for fifty years. In a startup you compress all this stress into three or four years. You do tend to get a certain bulk discount if you buy the economy-size pain, but you can't evade the fundamental conservation law. If starting a startup were easy, everyone would do it.

Millions, not Billions

If $3 million a year seems high to some people, it will seem low to others. Three *million?* How do I get to be a billionaire, like Bill Gates?

So let's get Bill Gates out of the way right now. It's not a good idea to use famous rich people as examples, because the press only write about the very richest, and these tend to be outliers. Bill Gates is a smart, determined, and hardworking man, but you need more than that to make as much money as he has. You also need to be very lucky.

There is a large random factor in the success of any company. So the guys you end up reading about in the papers are the ones who are very smart, totally dedicated, *and* win the lottery. Certainly Bill is smart and dedicated, but Microsoft also happens to have been the beneficiary of one of the most spectacular blunders in the history of business: the licensing deal for DOS. No doubt Bill did everything he could to steer IBM into making that blunder, and he has done an excellent job of exploiting it, but if there had been one person with a brain on IBM's side, Microsoft's future would have been very different. Microsoft at that stage had little leverage over IBM. They were effectively a component supplier. If IBM had required an exclusive license, as they should have, Microsoft would still have signed the deal. It would still have meant a lot of money for them, and IBM could easily have gotten an operating system elsewhere.

Instead IBM ended up using all its power in the market to give Microsoft control of the PC standard. From that point, all Microsoft had to do was execute. They never had to bet the company on a bold decision. All they had to do was play hardball with licensees and copy more innovative products reasonably promptly.

If IBM hadn't made this mistake, Microsoft would still have been a successful company, but it could not have grown so big so fast. Bill Gates would be rich, but he'd be somewhere near the bottom of the Forbes 400 with the other guys his age.

There are a lot of ways to get rich, and this essay is about only one of them. This essay is about how to make money by creating wealth and getting paid for it. There are plenty of other ways to get money, including chance, speculation, marriage, inheritance, theft, extortion, fraud, monopoly, graft, lobbying, counterfeiting, and prospecting. Most of the greatest fortunes have probably involved several of these.

The advantage of creating wealth, as a way to get rich, is not just that it's more legitimate (many of the other methods are now illegal) but that it's more *straightforward.* You just have to do something people want.

Money Is Not Wealth

If you want to create wealth, it will help to understand what it is. Wealth is not the same thing as money. [3] Wealth is as old as human history. Far older, in fact; ants have wealth. Money is a comparatively recent invention.

Wealth is the fundamental thing. Wealth is stuff we want: food, clothes, houses, cars, gadgets, travel to interesting places, and so on. You can have wealth without having money. If you had a magic machine that could on command make you a car or cook you dinner or do your laundry, or do anything else you wanted, you wouldn't need money. Whereas if you were in the middle of Antarctica, where there is nothing to buy, it wouldn't matter how much money you had.

Wealth is what you want, not money. But if wealth is the important thing, why does everyone talk about making money? It is a kind of shorthand: money is a way of moving wealth, and in practice they are usually interchangeable. But they are not the same thing, and unless you plan to get rich by counterfeiting, talking about *making money* can make it harder to understand how to make money.

Money is a side effect of specialization. In a specialized society, most of the things you need, you can't make for yourself. If you want a potato or a pencil or a place to live, you have to get it from someone else.

How do you get the person who grows the potatoes to give you some? By giving him something he wants in return. But you can't get very far by trading things directly with the people who need them. If you make violins, and none of the local farmers wants one, how will you eat?

The solution societies find, as they get more specialized, is to make the trade into a two-step process. Instead of trading violins directly for potatoes, you trade violins for, say, silver, which you can then trade again for anything else you need. The intermediate stuff-- the *medium of exchange*-- can be anything that's rare and portable. Historically metals have been the most common, but recently we've been using a medium of exchange, called the *dollar*, that doesn't physically exist. It works as a medium of exchange, however, because its rarity is guaranteed by the U.S. Government.

The advantage of a medium of exchange is that it makes trade work. The disadvantage is that it tends to obscure what trade really means. People think that what a business does is make money. But money is just the intermediate stage-- just a shorthand-- for whatever people want. What most businesses really do is make wealth. They do something people want. [4]

The Pie Fallacy

A surprising number of people retain from childhood the idea that there is a fixed amount of wealth in the world. There is, in any normal family, a fixed amount of *money* at any moment. But that's not the same thing.

When wealth is talked about in this context, it is often described as a pie. "You can't make the pie larger," say politicians. When you're talking about the amount of money in one family's bank account, or the amount available to a government from one year's tax revenue, this is true. If one person gets more, someone else has to get less.

I can remember believing, as a child, that if a few rich people had all the money, it left less for everyone else. Many people seem to continue to believe something like this well into adulthood. This fallacy is usually there in the background when you hear someone talking about how x percent of the population have y percent of the wealth. If you plan to start a startup, then whether you realize it or not, you're planning to disprove the Pie Fallacy.

What leads people astray here is the abstraction of money. Money is not wealth. It's just something we use to move wealth around. So although there may be, in certain specific moments (like your family, this month) a fixed amount of money available to trade with other people for things you want, there is not a fixed amount of wealth in the world. *You can make more wealth.* Wealth has been getting created and destroyed (but on balance, created) for all of human history.

Suppose you own a beat-up old car. Instead of sitting on your butt next summer, you could spend the time restoring your car to pristine condition. In doing so you create wealth. The world is-- and you specifically are-- one pristine old car the richer. And not just in some metaphorical way. If you sell your car, you'll get more for it.

In restoring your old car you have made yourself richer. You haven't made anyone

else poorer. So there is obviously not a fixed pie. And in fact, when you look at it this way, you wonder why anyone would think there was. [5]

Kids know, without knowing they know, that they can create wealth. If you need to give someone a present and don't have any money, you make one. But kids are so bad at making things that they consider home-made presents to be a distinct, inferior, sort of thing to store-bought ones-- a mere expression of the proverbial thought that counts. And indeed, the lumpy ashtrays we made for our parents did not have much of a resale market.

Craftsmen

The people most likely to grasp that wealth can be created are the ones who are good at making things, the craftsmen. Their hand-made objects become store-bought ones. But with the rise of industrialization there are fewer and fewer craftsmen. One of the biggest remaining groups is computer programmers.

A programmer can sit down in front of a computer and *create wealth*. A good piece of software is, in itself, a valuable thing. There is no manufacturing to confuse the issue. Those characters you type are a complete, finished product. If someone sat down and wrote a web browser that didn't suck (a fine idea, by the way), the world would be that much richer. [5b]

Everyone in a company works together to create wealth, in the sense of making more things people want. Many of the employees (e.g. the people in the mailroom or the personnel department) work at one remove from the actual making of stuff. Not the programmers. They literally think the product, one line at a time. And so it's clearer to programmers that wealth is something that's made, rather than being distributed, like slices of a pie, by some imaginary Daddy.

It's also obvious to programmers that there are huge variations in the rate at which wealth is created. At Viaweb we had one programmer who was a sort of monster of productivity. I remember watching what he did one long day and estimating that he had added several hundred thousand dollars to the market value of the company. A great programmer, on a roll, could create a million dollars worth of wealth in a couple weeks. A mediocre programmer over the same period will generate zero or even negative wealth (e.g. by introducing bugs).

This is why so many of the best programmers are libertarians. In our world, you sink or swim, and there are no excuses. When those far removed from the creation of wealth-- undergraduates, reporters, politicians-- hear that the richest 5% of the people have half the total wealth, they tend to think *injustice!* An experienced programmer would be more likely to think *is that all?* The top 5% of programmers probably write 99% of the good software.

Wealth can be created without being sold. Scientists, till recently at least, effectively donated the wealth they created. We are all richer for knowing about penicillin, because we're less likely to die from infections. Wealth is whatever

people want, and not dying is certainly something we want. Hackers often donate their work by writing open source software that anyone can use for free. I am much the richer for the operating system FreeBSD, which I'm running on the computer I'm using now, and so is Yahoo, which runs it on all their servers.

What a Job Is

In industrialized countries, people belong to one institution or another at least until their twenties. After all those years you get used to the idea of belonging to a group of people who all get up in the morning, go to some set of buildings, and do things that they do not, ordinarily, enjoy doing. Belonging to such a group becomes part of your identity: name, age, role, institution. If you have to introduce yourself, or someone else describes you, it will be as something like, John Smith, age 10, a student at such and such elementary school, or John Smith, age 20, a student at such and such college.

When John Smith finishes school he is expected to get a job. And what getting a job seems to mean is joining another institution. Superficially it's a lot like college. You pick the companies you want to work for and apply to join them. If one likes you, you become a member of this new group. You get up in the morning and go to a new set of buildings, and do things that you do not, ordinarily, enjoy doing. There are a few differences: life is not as much fun, and you get paid, instead of paying, as you did in college. But the similarities feel greater than the differences. John Smith is now John Smith, 22, a software developer at such and such corporation.

In fact John Smith's life has changed more than he realizes. Socially, a company looks much like college, but the deeper you go into the underlying reality, the more different it gets.

What a company does, and has to do if it wants to continue to exist, is earn money. And the way most companies make money is by creating wealth. Companies can be so specialized that this similarity is concealed, but it is not only manufacturing companies that create wealth. A big component of wealth is location. Remember that magic machine that could make you cars and cook you dinner and so on? It would not be so useful if it delivered your dinner to a random location in central Asia. If wealth means what people want, companies that move things also create wealth. Ditto for many other kinds of companies that don't make anything physical. Nearly all companies exist to do something people want.

And that's what you do, as well, when you go to work for a company. But here there is another layer that tends to obscure the underlying reality. In a company, the work you do is averaged together with a lot of other people's. You may not even be aware you're doing something people want. Your contribution may be indirect. But the company as a whole must be giving people something they want, or they won't make any money. And if they are paying you x dollars a year, then on average you must be contributing at least x dollars a year worth of work, or the company will be spending more than it makes, and will go out of business.

Someone graduating from college thinks, and is told, that he needs to get a job, as if the important thing were becoming a member of an institution. A more direct way to put it would be: you need to start doing something people want. You don't need to join a company to do that. All a company is is a group of people working together to do something people want. It's doing something people want that matters, not joining the group. [6]

For most people the best plan probably is to go to work for some existing company. But it is a good idea to understand what's happening when you do this. A job means doing something people want, averaged together with everyone else in that company.

Working Harder

That averaging gets to be a problem. I think the single biggest problem afflicting large companies is the difficulty of assigning a value to each person's work. For the most part they punt. In a big company you get paid a fairly predictable salary for working fairly hard. You're expected not to be obviously incompetent or lazy, but you're not expected to devote your whole life to your work.

It turns out, though, that there are economies of scale in how much of your life you devote to your work. In the right kind of business, someone who really devoted himself to work could generate ten or even a hundred times as much wealth as an average employee. A programmer, for example, instead of chugging along maintaining and updating an existing piece of software, could write a whole new piece of software, and with it create a new source of revenue.

Companies are not set up to reward people who want to do this. You can't go to your boss and say, I'd like to start working ten times as hard, so will you please pay me ten times as much? For one thing, the official fiction is that you are already working as hard as you can. But a more serious problem is that the company has no way of measuring the value of your work.

Salesmen are an exception. It's easy to measure how much revenue they generate, and they're usually paid a percentage of it. If a salesman wants to work harder, he can just start doing it, and he will automatically get paid proportionally more.

There is one other job besides sales where big companies can hire first-rate people: in the top management jobs. And for the same reason: their performance can be measured. The top managers are held responsible for the performance of the entire company. Because an ordinary employee's performance can't usually be measured, he is not expected to do more than put in a solid effort. Whereas top management, like salespeople, have to actually come up with the numbers. The CEO of a company that tanks cannot plead that he put in a solid effort. If the company does badly, he's done badly.

A company that could pay all its employees so straightforwardly would be enormously successful. Many employees would work harder if they could get paid for it. More importantly, such a company would attract people who wanted to work especially hard. It would crush its competitors.

Unfortunately, companies can't pay everyone like salesmen. Salesmen work alone. Most employees' work is tangled together. Suppose a company makes some kind of consumer gadget. The engineers build a reliable gadget with all kinds of new features; the industrial designers design a beautiful case for it; and then the marketing people convince everyone that it's something they've got to have. How do you know how much of the gadget's sales are due to each group's efforts? Or, for that matter, how much is due to the creators of past gadgets that gave the company a reputation for quality? There's no way to untangle all their contributions. Even if you could read the minds of the consumers, you'd find these factors were all blurred together.

If you want to go faster, it's a problem to have your work tangled together with a large number of other people's. In a large group, your performance is not separately measurable-- and the rest of the group slows you down.

Measurement and Leverage

To get rich you need to get yourself in a situation with two things, measurement and leverage. You need to be in a position where your performance can be measured, or there is no way to get paid more by doing more. And you have to have leverage, in the sense that the decisions you make have a big effect.

Measurement alone is not enough. An example of a job with measurement but not leverage is doing piecework in a sweatshop. Your performance is measured and you get paid accordingly, but you have no scope for decisions. The only decision you get to make is how fast you work, and that can probably only increase your earnings by a factor of two or three.

An example of a job with both measurement and leverage would be lead actor in a movie. Your performance can be measured in the gross of the movie. And you have leverage in the sense that your performance can make or break it.

CEOs also have both measurement and leverage. They're measured, in that the performance of the company is their performance. And they have leverage in that their decisions set the whole company moving in one direction or another.

I think everyone who gets rich by their own efforts will be found to be in a situation with measurement and leverage. Everyone I can think of does: CEOs, movie stars, hedge fund managers, professional athletes. A good hint to the presence of leverage is the possibility of failure. Upside must be balanced by downside, so if there is big potential for gain there must also be a terrifying possibility of loss. CEOs, stars, fund managers, and athletes all live with the sword hanging over their heads; the moment they start to suck, they're out. If you're in a

job that feels safe, you are not going to get rich, because if there is no danger there is almost certainly no leverage.

But you don't have to become a CEO or a movie star to be in a situation with measurement and leverage. All you need to do is be part of a small group working on a hard problem.

Smallness = Measurement

If you can't measure the value of the work done by individual employees, you can get close. You can measure the value of the work done by small groups.

One level at which you can accurately measure the revenue generated by employees is at the level of the whole company. When the company is small, you are thereby fairly close to measuring the contributions of individual employees. A viable startup might only have ten employees, which puts you within a factor of ten of measuring individual effort.

Starting or joining a startup is thus as close as most people can get to saying to one's boss, I want to work ten times as hard, so please pay me ten times as much. There are two differences: you're not saying it to your boss, but directly to the customers (for whom your boss is only a proxy after all), and you're not doing it individually, but along with a small group of other ambitious people.

It will, ordinarily, be a group. Except in a few unusual kinds of work, like acting or writing books, you can't be a company of one person. And the people you work with had better be good, because it's their work that yours is going to be averaged with.

A big company is like a giant galley driven by a thousand rowers. Two things keep the speed of the galley down. One is that individual rowers don't see any result from working harder. The other is that, in a group of a thousand people, the average rower is likely to be pretty average.

If you took ten people at random out of the big galley and put them in a boat by themselves, they could probably go faster. They would have both carrot and stick to motivate them. An energetic rower would be encouraged by the thought that he could have a visible effect on the speed of the boat. And if someone was lazy, the others would be more likely to notice and complain.

But the real advantage of the ten-man boat shows when you take the ten *best* rowers out of the big galley and put them in a boat together. They will have all the extra motivation that comes from being in a small group. But more importantly, by selecting that small a group you can get the best rowers. Each one will be in the top 1%. It's a much better deal for them to average their work together with a small group of their peers than to average it with everyone.

That's the real point of startups. Ideally, you are getting together with a group of

other people who also want to work a lot harder, and get paid a lot more, than they would in a big company. And because startups tend to get founded by self-selecting groups of ambitious people who already know one another (at least by reputation), the level of measurement is more precise than you get from smallness alone. A startup is not merely ten people, but ten people like you.

Steve Jobs once said that the success or failure of a startup depends on the first ten employees. I agree. If anything, it's more like the first five. Being small is not, in itself, what makes startups kick butt, but rather that small groups can be select. You don't want small in the sense of a village, but small in the sense of an all-star team.

The larger a group, the closer its average member will be to the average for the population as a whole. So all other things being equal, a very able person in a big company is probably getting a bad deal, because his performance is dragged down by the overall lower performance of the others. Of course, all other things often are not equal: the able person may not care about money, or may prefer the stability of a large company. But a very able person who does care about money will ordinarily do better to go off and work with a small group of peers.

Technology = Leverage

Startups offer anyone a way to be in a situation with measurement and leverage. They allow measurement because they're small, and they offer leverage because they make money by inventing new technology.

What is technology? It's *technique*. It's the way we all do things. And when you discover a new way to do things, its value is multiplied by all the people who use it. It is the proverbial fishing rod, rather than the fish. That's the difference between a startup and a restaurant or a barber shop. You fry eggs or cut hair one customer at a time. Whereas if you solve a technical problem that a lot of people care about, you help everyone who uses your solution. That's leverage.

If you look at history, it seems that most people who got rich by creating wealth did it by developing new technology. You just can't fry eggs or cut hair fast enough. What made the Florentines rich in 1200 was the discovery of new techniques for making the high-tech product of the time, fine woven cloth. What made the Dutch rich in 1600 was the discovery of shipbuilding and navigation techniques that enabled them to dominate the seas of the Far East.

Fortunately there is a natural fit between smallness and solving hard problems. The leading edge of technology moves fast. Technology that's valuable today could be worthless in a couple years. Small companies are more at home in this world, because they don't have layers of bureaucracy to slow them down. Also, technical advances tend to come from unorthodox approaches, and small companies are less constrained by convention.

Big companies can develop technology. They just can't do it quickly. Their size

makes them slow and prevents them from rewarding employees for the extraordinary effort required. So in practice big companies only get to develop technology in fields where large capital requirements prevent startups from competing with them, like microprocessors, power plants, or passenger aircraft. And even in those fields they depend heavily on startups for components and ideas.

It's obvious that biotech or software startups exist to solve hard technical problems, but I think it will also be found to be true in businesses that don't seem to be about technology. McDonald's, for example, grew big by designing a system, the McDonald's franchise, that could then be reproduced at will all over the face of the earth. A McDonald's franchise is controlled by rules so precise that it is practically a piece of software. Write once, run everywhere. Ditto for Wal-Mart. Sam Walton got rich not by being a retailer, but by designing a new kind of store.

Use difficulty as a guide not just in selecting the overall aim of your company, but also at decision points along the way. At Viaweb one of our rules of thumb was *run upstairs.* Suppose you are a little, nimble guy being chased by a big, fat, bully. You open a door and find yourself in a staircase. Do you go up or down? I say up. The bully can probably run downstairs as fast as you can. Going upstairs his bulk will be more of a disadvantage. Running upstairs is hard for you but even harder for him.

What this meant in practice was that we deliberately sought hard problems. If there were two features we could add to our software, both equally valuable in proportion to their difficulty, we'd always take the harder one. Not just because it was more valuable, but *because it was harder.* We delighted in forcing bigger, slower competitors to follow us over difficult ground. Like guerillas, startups prefer the difficult terrain of the mountains, where the troops of the central government can't follow. I can remember times when we were just exhausted after wrestling all day with some horrible technical problem. And I'd be delighted, because something that was hard for us would be impossible for our competitors.

This is not just a good way to run a startup. It's what a startup is. Venture capitalists know about this and have a phrase for it: *barriers to entry.* If you go to a VC with a new idea and ask him to invest in it, one of the first things he'll ask is, how hard would this be for someone else to develop? That is, how much difficult ground have you put between yourself and potential pursuers? [7] And you had better have a convincing explanation of why your technology would be hard to duplicate. Otherwise as soon as some big company becomes aware of it, they'll make their own, and with their brand name, capital, and distribution clout, they'll take away your market overnight. You'd be like guerillas caught in the open field by regular army forces.

One way to put up barriers to entry is through patents. But patents may not provide much protection. Competitors commonly find ways to work around a patent. And if they can't, they may simply violate it and invite you to sue them. A big company is not afraid to be sued; it's an everyday thing for them. They'll make

sure that suing them is expensive and takes a long time. Ever heard of Philo Farnsworth? He invented television. The reason you've never heard of him is that his company was not the one to make money from it. [8] The company that did was RCA, and Farnsworth's reward for his efforts was a decade of patent litigation.

Here, as so often, the best defense is a good offense. If you can develop technology that's simply too hard for competitors to duplicate, you don't need to rely on other defenses. Start by picking a hard problem, and then at every decision point, take the harder choice. [9]

The Catch(es)

If it were simply a matter of working harder than an ordinary employee and getting paid proportionately, it would obviously be a good deal to start a startup. Up to a point it would be more fun. I don't think many people like the slow pace of big companies, the interminable meetings, the water-cooler conversations, the clueless middle managers, and so on.

Unfortunately there are a couple catches. One is that you can't choose the point on the curve that you want to inhabit. You can't decide, for example, that you'd like to work just two or three times as hard, and get paid that much more. When you're running a startup, your competitors decide how hard you work. And they pretty much all make the same decision: as hard as you possibly can.

The other catch is that the payoff is only on average proportionate to your productivity. There is, as I said before, a large random multiplier in the success of any company. So in practice the deal is not that you're 30 times as productive and get paid 30 times as much. It is that you're 30 times as productive, and get paid between zero and a thousand times as much. If the mean is 30x, the median is probably zero. Most startups tank, and not just the dogfood portals we all heard about during the Internet Bubble. It's common for a startup to be developing a genuinely good product, take slightly too long to do it, run out of money, and have to shut down.

A startup is like a mosquito. A bear can absorb a hit and a crab is armored against one, but a mosquito is designed for one thing: to score. No energy is wasted on defense. The defense of mosquitos, as a species, is that there are a lot of them, but this is little consolation to the individual mosquito.

Startups, like mosquitos, tend to be an all-or-nothing proposition. And you don't generally know which of the two you're going to get till the last minute. Viaweb came close to tanking several times. Our trajectory was like a sine wave. Fortunately we got bought at the top of the cycle, but it was damned close. While we were visiting Yahoo in California to talk about selling the company to them, we had to borrow a conference room to reassure an investor who was about to back out of a new round of funding that we needed to stay alive.

The all-or-nothing aspect of startups was not something we wanted. Viaweb's

hackers were all extremely risk-averse. If there had been some way just to work super hard and get paid for it, without having a lottery mixed in, we would have been delighted. We would have much preferred a 100% chance of $1 million to a 20% chance of $10 million, even though theoretically the second is worth twice as much. Unfortunately, there is not currently any space in the business world where you can get the first deal.

The closest you can get is by selling your startup in the early stages, giving up upside (and risk) for a smaller but guaranteed payoff. We had a chance to do this, and stupidly, as we then thought, let it slip by. After that we became comically eager to sell. For the next year or so, if anyone expressed the slightest curiosity about Viaweb we would try to sell them the company. But there were no takers, so we had to keep going.

It would have been a bargain to buy us at an early stage, but companies doing acquisitions are not looking for bargains. A company big enough to acquire startups will be big enough to be fairly conservative, and within the company the people in charge of acquisitions will be among the more conservative, because they are likely to be business school types who joined the company late. They would rather overpay for a safe choice. So it is easier to sell an established startup, even at a large premium, than an early-stage one.

Get Users

I think it's a good idea to get bought, if you can. Running a business is different from growing one. It is just as well to let a big company take over once you reach cruising altitude. It's also financially wiser, because selling allows you to diversify. What would you think of a financial advisor who put all his client's assets into one volatile stock?

How do you get bought? Mostly by doing the same things you'd do if you didn't intend to sell the company. Being profitable, for example. But getting bought is also an art in its own right, and one that we spent a lot of time trying to master.

Potential buyers will always delay if they can. The hard part about getting bought is getting them to act. For most people, the most powerful motivator is not the hope of gain, but the fear of loss. For potential acquirers, the most powerful motivator is the prospect that one of their competitors will buy you. This, as we found, causes CEOs to take red-eyes. The second biggest is the worry that, if they don't buy you now, you'll continue to grow rapidly and will cost more to acquire later, or even become a competitor.

In both cases, what it all comes down to is users. You'd think that a company about to buy you would do a lot of research and decide for themselves how valuable your technology was. Not at all. What they go by is the number of users you have.

In effect, acquirers assume the customers know who has the best technology. And

this is not as stupid as it sounds. Users are the only real proof that you've created wealth. Wealth is what people want, and if people aren't using your software, maybe it's not just because you're bad at marketing. Maybe it's because you haven't made what they want.

Venture capitalists have a list of danger signs to watch out for. Near the top is the company run by techno-weenies who are obsessed with solving interesting technical problems, instead of making users happy. In a startup, you're not just trying to solve problems. You're trying to solve problems *that users care about.*

So I think you should make users the test, just as acquirers do. Treat a startup as an optimization problem in which performance is measured by number of users. As anyone who has tried to optimize software knows, the key is measurement. When you try to guess where your program is slow, and what would make it faster, you almost always guess wrong.

Number of users may not be the perfect test, but it will be very close. It's what acquirers care about. It's what revenues depend on. It's what makes competitors unhappy. It's what impresses reporters, and potential new users. Certainly it's a better test than your a priori notions of what problems are important to solve, no matter how technically adept you are.

Among other things, treating a startup as an optimization problem will help you avoid another pitfall that VCs worry about, and rightly-- taking a long time to develop a product. Now we can recognize this as something hackers already know to avoid: premature optimization. Get a version 1.0 out there as soon as you can. Until you have some users to measure, you're optimizing based on guesses.

The ball you need to keep your eye on here is the underlying principle that wealth is what people want. If you plan to get rich by creating wealth, you have to know what people want. So few businesses really pay attention to making customers happy. How often do you walk into a store, or call a company on the phone, with a feeling of dread in the back of your mind? When you hear "your call is important to us, please stay on the line," do you think, oh good, now everything will be all right?

A restaurant can afford to serve the occasional burnt dinner. But in technology, you cook one thing and that's what everyone eats. So any difference between what people want and what you deliver is multiplied. You please or annoy customers wholesale. The closer you can get to what they want, the more wealth you generate.

Wealth and Power

Making wealth is not the only way to get rich. For most of human history it has not even been the most common. Until a few centuries ago, the main sources of wealth were mines, slaves and serfs, land, and cattle, and the only ways to acquire these rapidly were by inheritance, marriage, conquest, or confiscation. Naturally

wealth had a bad reputation.

Two things changed. The first was the rule of law. For most of the world's history, if you did somehow accumulate a fortune, the ruler or his henchmen would find a way to steal it. But in medieval Europe something new happened. A new class of merchants and manufacturers began to collect in towns. [10] Together they were able to withstand the local feudal lord. So for the first time in our history, the bullies stopped stealing the nerds' lunch money. This was naturally a great incentive, and possibly indeed the main cause of the second big change, industrialization.

A great deal has been written about the causes of the Industrial Revolution. But surely a necessary, if not sufficient, condition was that people who made fortunes be able to enjoy them in peace. [11] One piece of evidence is what happened to countries that tried to return to the old model, like the Soviet Union, and to a lesser extent Britain under the labor governments of the 1960s and early 1970s. Take away the incentive of wealth, and technical innovation grinds to a halt.

Remember what a startup is, economically: a way of saying, I want to work faster. Instead of accumulating money slowly by being paid a regular wage for fifty years, I want to get it over with as soon as possible. So governments that forbid you to accumulate wealth are in effect decreeing that you work slowly. They're willing to let you earn $3 million over fifty years, but they're not willing to let you work so hard that you can do it in two. They are like the corporate boss that you can't go to and say, I want to work ten times as hard, so please pay me ten times a much. Except this is not a boss you can escape by starting your own company.

The problem with working slowly is not just that technical innovation happens slowly. It's that it tends not to happen at all. It's only when you're deliberately looking for hard problems, as a way to use speed to the greatest advantage, that you take on this kind of project. Developing new technology is a pain in the ass. It is, as Edison said, one percent inspiration and ninety-nine percent perspiration. Without the incentive of wealth, no one wants to do it. Engineers will work on sexy projects like fighter planes and moon rockets for ordinary salaries, but more mundane technologies like light bulbs or semiconductors have to be developed by entrepreneurs.

Startups are not just something that happened in Silicon Valley in the last couple decades. Since it became possible to get rich by creating wealth, everyone who has done it has used essentially the same recipe: measurement and leverage, where measurement comes from working with a small group, and leverage from developing new techniques. The recipe was the same in Florence in 1200 as it is in Santa Clara today.

Understanding this may help to answer an important question: why Europe grew so powerful. Was it something about the geography of Europe? Was it that Europeans are somehow racially superior? Was it their religion? The answer (or at least the proximate cause) may be that the Europeans rode on the crest of a

powerful new idea: allowing those who made a lot of money to keep it.

Once you're allowed to do that, people who want to get rich can do it by generating wealth instead of stealing it. The resulting technological growth translates not only into wealth but into military power. The theory that led to the stealth plane was developed by a Soviet mathematician. But because the Soviet Union didn't have a computer industry, it remained for them a theory; they didn't have hardware capable of executing the calculations fast enough to design an actual airplane.

In that respect the Cold War teaches the same lesson as World War II and, for that matter, most wars in recent history. Don't let a ruling class of warriors and politicians squash the entrepreneurs. The same recipe that makes individuals rich makes countries powerful. Let the nerds keep their lunch money, and you rule the world.

Notes

[1] One valuable thing you tend to get only in startups is *uninterruptability*. Different kinds of work have different time quanta. Someone proofreading a manuscript could probably be interrupted every fifteen minutes with little loss of productivity. But the time quantum for hacking is very long: it might take an hour just to load a problem into your head. So the cost of having someone from personnel call you about a form you forgot to fill out can be huge.

This is why hackers give you such a baleful stare as they turn from their screen to answer your question. Inside their heads a giant house of cards is tottering.

The mere possibility of being interrupted deters hackers from starting hard projects. This is why they tend to work late at night, and why it's next to impossible to write great software in a cubicle (except late at night).

One great advantage of startups is that they don't yet have any of the people who interrupt you. There is no personnel department, and thus no form nor anyone to call you about it.

[2] Faced with the idea that people working for startups might be 20 or 30 times as productive as those working for large companies, executives at large companies will naturally wonder, how could I get the people working for me to do that? The answer is simple: pay them to.

Internally most companies are run like Communist states. If you believe in free markets, why not turn your company into one?

Hypothesis: A company will be maximally profitable when each employee is paid in proportion to the wealth they generate.

[3] Until recently even governments sometimes didn't grasp the distinction between money and wealth. Adam Smith (*Wealth of Nations*, v:i) mentions several that tried to preserve their "wealth" by forbidding the export of gold or silver. But having more of the medium of exchange would not make a country richer; if you have more money chasing the same amount of material wealth, the only result is higher prices.

[4] There are many senses of the word "wealth," not all of them material. I'm not trying to make a deep philosophical point here about which is the true kind. I'm writing about one specific, rather technical sense of the word "wealth." What people will give you money for. This is an interesting sort of wealth to study, because it is the kind that prevents you from starving. And what people will give you money for depends on them, not you.

When you're starting a business, it's easy to slide into thinking that customers want what you do. During the Internet Bubble I talked to a woman who, because she liked the outdoors, was starting an "outdoor portal." You know what kind of business you should start if you like the outdoors? One to recover data from crashed hard disks.

What's the connection? None at all. Which is precisely my point. If you want to create wealth (in the narrow technical sense of not starving) then you should be especially skeptical about any plan that centers on things you like doing. That is where your idea of what's valuable is least likely to coincide with other people's.

[5] In the average car restoration you probably do make everyone else microscopically poorer, by doing a small amount of damage to the environment. While environmental costs should be taken into account, they don't make wealth a zero-sum game. For example, if you repair a machine that's broken because a part has come unscrewed, you create wealth with no environmental cost.

[5b] This essay was written before Firefox.

[6] Many people feel confused and depressed in their early twenties. Life seemed so much more fun in college. Well, of course it was. Don't be fooled by the surface similarities. You've gone from guest to servant. It's possible to have fun in this new world. Among other things, you now get to go behind the doors that say "authorized personnel only." But the change is a shock at first, and all the worse if you're not consciously aware of it.

[7] When VCs asked us how long it would take another startup to duplicate our software, we used to reply that they probably wouldn't be able to at all. I think this made us seem naive, or liars.

[8] Few technologies have one clear inventor. So as a rule, if you know the "inventor" of something (the telephone, the assembly line, the airplane, the light bulb, the transistor) it is because their company made money from it, and the

company's PR people worked hard to spread the story. If you don't know who invented something (the automobile, the television, the computer, the jet engine, the laser), it's because other companies made all the money.

[9] This is a good plan for life in general. If you have two choices, choose the harder. If you're trying to decide whether to go out running or sit home and watch TV, go running. Probably the reason this trick works so well is that when you have two choices and one is harder, the only reason you're even considering the other is laziness. You know in the back of your mind what's the right thing to do, and this trick merely forces you to acknowledge it.

[10] It is probably no accident that the middle class first appeared in northern Italy and the low countries, where there were no strong central governments. These two regions were the richest of their time and became the twin centers from which Renaissance civilization radiated. If they no longer play that role, it is because other places, like the United States, have been truer to the principles they discovered.

[11] It may indeed be a sufficient condition. But if so, why didn't the Industrial Revolution happen earlier? Two possible (and not incompatible) answers: (a) It did. The Industrial Revolution was one in a series. (b) Because in medieval towns, monopolies and guild regulations initially slowed the development of new means of production.

[Comment](#) on this essay.

[Russian Translation](#)

·

[Arabic Translation](#)

·

·

You'll find this essay and 14 others in ***Hackers & Painters***.

Mind the Gap

May 2004

When people care enough about something to do it well, those who do it best tend to be far better than everyone else. There's a huge gap between Leonardo and second-rate contemporaries like Borgognone. You see the same gap between Raymond Chandler and the average writer of detective novels. A top-ranked professional chess player could play ten thousand games against an ordinary club player without losing once.

Like chess or painting or writing novels, making money is a very specialized skill. But for some reason we treat this skill differently. No one complains when a few people surpass all the rest at playing chess or writing novels, but when a few people make more money than the rest, we get editorials saying this is wrong.

Why? The pattern of variation seems no different than for any other skill. What causes people to react so strongly when the skill is making money?

I think there are three reasons we treat making money as different: the misleading model of wealth we learn as children; the disreputable way in which, till recently, most fortunes were accumulated; and the worry that great variations in income are somehow bad for society. As far as I can tell, the first is mistaken, the second outdated, and the third empirically false. Could it be that, in a modern democracy, variation in income is actually a sign of health?

The Daddy Model of Wealth

When I was five I thought electricity was created by electric sockets. I didn't realize there were power plants out there generating it. Likewise, it doesn't occur to most kids that wealth is something that has to be generated. It seems to be something that flows from parents.

Because of the circumstances in which they encounter it, children tend to misunderstand wealth. They confuse it with money. They think that there is a fixed amount of it. And they think of it as something that's distributed by authorities (and so should be distributed equally), rather than something that has to be created (and might be created unequally).

In fact, wealth is not money. Money is just a convenient way of trading one form of

wealth for another. Wealth is the underlying stuff—the goods and services we buy. When you travel to a rich or poor country, you don't have to look at people's bank accounts to tell which kind you're in. You can *see* wealth—in buildings and streets, in the clothes and the health of the people.

Where does wealth come from? People make it. This was easier to grasp when most people lived on farms, and made many of the things they wanted with their own hands. Then you could see in the house, the herds, and the granary the wealth that each family created. It was obvious then too that the wealth of the world was not a fixed quantity that had to be shared out, like slices of a pie. If you wanted more wealth, you could make it.

This is just as true today, though few of us create wealth directly for ourselves (except for a few vestigial domestic tasks). Mostly we create wealth for other people in exchange for money, which we then trade for the forms of wealth we want. [1]

Because kids are unable to create wealth, whatever they have has to be given to them. And when wealth is something you're given, then of course it seems that it should be distributed equally. [2] As in most families it is. The kids see to that. "Unfair," they cry, when one sibling gets more than another.

In the real world, you can't keep living off your parents. If you want something, you either have to make it, or do something of equivalent value for someone else, in order to get them to give you enough money to buy it. In the real world, wealth is (except for a few specialists like thieves and speculators) something you have to create, not something that's distributed by Daddy. And since the ability and desire to create it vary from person to person, it's not made equally.

You get paid by doing or making something people want, and those who make more money are often simply better at doing what people want. Top actors make a lot more money than B-list actors. The B-list actors might be almost as charismatic, but when people go to the theater and look at the list of movies playing, they want that extra oomph that the big stars have.

Doing what people want is not the only way to get money, of course. You could also rob banks, or solicit bribes, or establish a monopoly. Such tricks account for some variation in wealth, and indeed for some of the biggest individual fortunes, but they are not the root cause of variation in income. The root cause of variation in income, as Occam's Razor implies, is the same as the root cause of variation in every other human skill.

In the United States, the CEO of a large public company makes about 100 times as much as the average person. [3] Basketball players make about 128 times as much, and baseball players 72 times as much. Editorials quote this kind of statistic with horror. But I have no trouble imagining that one person could be 100 times as productive as another. In ancient Rome the price of *slaves* varied by a factor of 50 depending on their skills. [4] And that's without considering motivation, or the

extra leverage in productivity that you can get from modern technology.

Editorials about athletes' or CEOs' salaries remind me of early Christian writers, arguing from first principles about whether the Earth was round, when they could just walk outside and check. [5] How much someone's work is worth is not a policy question. It's something the market already determines.

"Are they really worth 100 of us?" editorialists ask. Depends on what you mean by worth. If you mean worth in the sense of what people will pay for their skills, the answer is yes, apparently.

A few CEOs' incomes reflect some kind of wrongdoing. But are there not others whose incomes really do reflect the wealth they generate? Steve Jobs saved a company that was in a terminal decline. And not merely in the way a turnaround specialist does, by cutting costs; he had to decide what Apple's next products should be. Few others could have done it. And regardless of the case with CEOs, it's hard to see how anyone could argue that the salaries of professional basketball players don't reflect supply and demand.

It may seem unlikely in principle that one individual could really generate so much more wealth than another. The key to this mystery is to revisit that question, are they really worth 100 of us? *Would* a basketball team trade one of their players for 100 random people? What would Apple's next product look like if you replaced Steve Jobs with a committee of 100 random people? [6] These things don't scale linearly. Perhaps the CEO or the professional athlete has only ten times (whatever that means) the skill and determination of an ordinary person. But it makes all the difference that it's concentrated in one individual.

When we say that one kind of work is overpaid and another underpaid, what are we really saying? In a free market, prices are determined by what buyers want. People like baseball more than poetry, so baseball players make more than poets. To say that a certain kind of work is underpaid is thus identical with saying that people want the wrong things.

Well, of course people want the wrong things. It seems odd to be surprised by that. And it seems even odder to say that it's *unjust* that certain kinds of work are underpaid. [7] Then you're saying that it's unjust that people want the wrong things. It's lamentable that people prefer reality TV and corndogs to Shakespeare and steamed vegetables, but unjust? That seems like saying that blue is heavy, or that up is circular.

The appearance of the word "unjust" here is the unmistakable spectral signature of the Daddy Model. Why else would this idea occur in this odd context? Whereas if the speaker were still operating on the Daddy Model, and saw wealth as something that flowed from a common source and had to be shared out, rather than something generated by doing what other people wanted, this is exactly what you'd get on noticing that some people made much more than others.

When we talk about "unequal distribution of income," we should also ask, where does that income come from? [8] Who made the wealth it represents? Because to the extent that income varies simply according to how much wealth people create, the distribution may be unequal, but it's hardly unjust.

Stealing It

The second reason we tend to find great disparities of wealth alarming is that for most of human history the usual way to accumulate a fortune was to steal it: in pastoral societies by cattle raiding; in agricultural societies by appropriating others' estates in times of war, and taxing them in times of peace.

In conflicts, those on the winning side would receive the estates confiscated from the losers. In England in the 1060s, when William the Conqueror distributed the estates of the defeated Anglo-Saxon nobles to his followers, the conflict was military. By the 1530s, when Henry VIII distributed the estates of the monasteries to his followers, it was mostly political. [9] But the principle was the same. Indeed, the same principle is at work now in Zimbabwe.

In more organized societies, like China, the ruler and his officials used taxation instead of confiscation. But here too we see the same principle: the way to get rich was not to create wealth, but to serve a ruler powerful enough to appropriate it.

This started to change in Europe with the rise of the middle class. Now we think of the middle class as people who are neither rich nor poor, but originally they were a distinct group. In a feudal society, there are just two classes: a warrior aristocracy, and the serfs who work their estates. The middle class were a new, third group who lived in towns and supported themselves by manufacturing and trade.

Starting in the tenth and eleventh centuries, petty nobles and former serfs banded together in towns that gradually became powerful enough to ignore the local feudal lords. [10] Like serfs, the middle class made a living largely by creating wealth. (In port cities like Genoa and Pisa, they also engaged in piracy.) But unlike serfs they had an incentive to create a lot of it. Any wealth a serf created belonged to his master. There was not much point in making more than you could hide. Whereas the independence of the townsmen allowed them to keep whatever wealth they created.

Once it became possible to get rich by creating wealth, society as a whole started to get richer very rapidly. Nearly everything we have was created by the middle class. Indeed, the other two classes have effectively disappeared in industrial societies, and their names been given to either end of the middle class. (In the original sense of the word, Bill Gates is middle class.)

But it was not till the Industrial Revolution that wealth creation definitively replaced corruption as the best way to get rich. In England, at least, corruption only became unfashionable (and in fact only started to be called "corruption") when there started to be other, faster ways to get rich.

Seventeenth-century England was much like the third world today, in that government office was a recognized route to wealth. The great fortunes of that time still derived more from what we would now call corruption than from commerce. [11] By the nineteenth century that had changed. There continued to be bribes, as there still are everywhere, but politics had by then been left to men who were driven more by vanity than greed. Technology had made it possible to create wealth faster than you could steal it. The prototypical rich man of the nineteenth century was not a courtier but an industrialist.

With the rise of the middle class, wealth stopped being a zero-sum game. Jobs and Wozniak didn't have to make us poor to make themselves rich. Quite the opposite: they created things that made our lives materially richer. They had to, or we wouldn't have paid for them.

But since for most of the world's history the main route to wealth was to steal it, we tend to be suspicious of rich people. Idealistic undergraduates find their unconsciously preserved child's model of wealth confirmed by eminent writers of the past. It is a case of the mistaken meeting the outdated.

"Behind every great fortune, there is a crime," Balzac wrote. Except he didn't. What he actually said was that a great fortune with no apparent cause was probably due to a crime well enough executed that it had been forgotten. If we were talking about Europe in 1000, or most of the third world today, the standard misquotation would be spot on. But Balzac lived in nineteenth-century France, where the Industrial Revolution was well advanced. He knew you could make a fortune without stealing it. After all, he did himself, as a popular novelist. [12]

Only a few countries (by no coincidence, the richest ones) have reached this stage. In most, corruption still has the upper hand. In most, the fastest way to get wealth is by stealing it. And so when we see increasing differences in income in a rich country, there is a tendency to worry that it's sliding back toward becoming another Venezuela. I think the opposite is happening. I think you're seeing a country a full step ahead of Venezuela.

The Lever of Technology

Will technology increase the gap between rich and poor? It will certainly increase the gap between the productive and the unproductive. That's the whole point of technology. With a tractor an energetic farmer could plow six times as much land in a day as he could with a team of horses. But only if he mastered a new kind of farming.

I've seen the lever of technology grow visibly in my own time. In high school I made money by mowing lawns and scooping ice cream at Baskin-Robbins. This was the only kind of work available at the time. Now high school kids could write software or design web sites. But only some of them will; the rest will still be scooping ice cream.

I remember very vividly when in 1985 improved technology made it possible for me to buy a computer of my own. Within months I was using it to make money as a freelance programmer. A few years before, I couldn't have done this. A few years before, there was no such *thing* as a freelance programmer. But Apple created wealth, in the form of powerful, inexpensive computers, and programmers immediately set to work using it to create more.

As this example suggests, the rate at which technology increases our productive capacity is probably exponential, rather than linear. So we should expect to see ever-increasing variation in individual productivity as time goes on. Will that increase the gap between rich and the poor? Depends which gap you mean.

Technology should increase the gap in income, but it seems to decrease other gaps. A hundred years ago, the rich led a different *kind* of life from ordinary people. They lived in houses full of servants, wore elaborately uncomfortable clothes, and travelled about in carriages drawn by teams of horses which themselves required their own houses and servants. Now, thanks to technology, the rich live more like the average person.

Cars are a good example of why. It's possible to buy expensive, handmade cars that cost hundreds of thousands of dollars. But there is not much point. Companies make more money by building a large number of ordinary cars than a small number of expensive ones. So a company making a mass-produced car can afford to spend a lot more on its design. If you buy a custom-made car, something will always be breaking. The only point of buying one now is to advertise that you can.

Or consider watches. Fifty years ago, by spending a lot of money on a watch you could get better performance. When watches had mechanical movements, expensive watches kept better time. Not any more. Since the invention of the quartz movement, an ordinary Timex is more accurate than a Patek Philippe costing hundreds of thousands of dollars. [13] Indeed, as with expensive cars, if you're determined to spend a lot of money on a watch, you have to put up with some inconvenience to do it: as well as keeping worse time, mechanical watches have to be wound.

The only thing technology can't cheapen is brand. Which is precisely why we hear ever more about it. Brand is the residue left as the substantive differences between rich and poor evaporate. But what label you have on your stuff is a much smaller matter than having it versus not having it. In 1900, if you kept a carriage, no one asked what year or brand it was. If you had one, you were rich. And if you weren't rich, you took the omnibus or walked. Now even the poorest Americans drive cars, and it is only because we're so well trained by advertising that we can even recognize the especially expensive ones. [14]

The same pattern has played out in industry after industry. If there is enough demand for something, technology will make it cheap enough to sell in large volumes, and the mass-produced versions will be, if not better, at least more

convenient. [15] And there is nothing the rich like more than convenience. The rich people I know drive the same cars, wear the same clothes, have the same kind of furniture, and eat the same foods as my other friends. Their houses are in different neighborhoods, or if in the same neighborhood are different sizes, but within them life is similar. The houses are made using the same construction techniques and contain much the same objects. It's inconvenient to do something expensive and custom.

The rich spend their time more like everyone else too. Bertie Wooster seems long gone. Now, most people who are rich enough not to work do anyway. It's not just social pressure that makes them; idleness is lonely and demoralizing.

Nor do we have the social distinctions there were a hundred years ago. The novels and etiquette manuals of that period read now like descriptions of some strange tribal society. "With respect to the continuance of friendships..." hints *Mrs. Beeton's Book of Household Management* (1880), "it may be found necessary, in some cases, for a mistress to relinquish, on assuming the responsibility of a household, many of those commenced in the earlier part of her life." A woman who married a rich man was expected to drop friends who didn't. You'd seem a barbarian if you behaved that way today. You'd also have a very boring life. People still tend to segregate themselves somewhat, but much more on the basis of education than wealth. [16]

Materially and socially, technology seems to be decreasing the gap between the rich and the poor, not increasing it. If Lenin walked around the offices of a company like Yahoo or Intel or Cisco, he'd think communism had won. Everyone would be wearing the same clothes, have the same kind of office (or rather, cubicle) with the same furnishings, and address one another by their first names instead of by honorifics. Everything would seem exactly as he'd predicted, until he looked at their bank accounts. Oops.

Is it a problem if technology increases that gap? It doesn't seem to be so far. As it increases the gap in income, it seems to decrease most other gaps.

Alternative to an Axiom

One often hears a policy criticized on the grounds that it would increase the income gap between rich and poor. As if it were an axiom that this would be bad. It might be true that increased variation in income would be bad, but I don't see how we can say it's *axiomatic.*

Indeed, it may even be false, in industrial democracies. In a society of serfs and warlords, certainly, variation in income is a sign of an underlying problem. But serfdom is not the only cause of variation in income. A 747 pilot doesn't make 40 times as much as a checkout clerk because he is a warlord who somehow holds her in thrall. His skills are simply much more valuable.

I'd like to propose an alternative idea: that in a modern society, increasing

variation in income is a sign of health. Technology seems to increase the variation in productivity at faster than linear rates. If we don't see corresponding variation in income, there are three possible explanations: (a) that technical innovation has stopped, (b) that the people who would create the most wealth aren't doing it, or (c) that they aren't getting paid for it.

I think we can safely say that (a) and (b) would be bad. If you disagree, try living for a year using only the resources available to the average Frankish nobleman in 800, and report back to us. (I'll be generous and not send you back to the stone age.)

The only option, if you're going to have an increasingly prosperous society without increasing variation in income, seems to be (c), that people will create a lot of wealth without being paid for it. That Jobs and Wozniak, for example, will cheerfully work 20-hour days to produce the Apple computer for a society that allows them, after taxes, to keep just enough of their income to match what they would have made working 9 to 5 at a big company.

Will people create wealth if they can't get paid for it? Only if it's fun. People will write operating systems for free. But they won't install them, or take support calls, or train customers to use them. And at least 90% of the work that even the highest tech companies do is of this second, unedifying kind.

All the unfun kinds of wealth creation slow dramatically in a society that confiscates private fortunes. We can confirm this empirically. Suppose you hear a strange noise that you think may be due to a nearby fan. You turn the fan off, and the noise stops. You turn the fan back on, and the noise starts again. Off, quiet. On, noise. In the absence of other information, it would seem the noise is caused by the fan.

At various times and places in history, whether you could accumulate a fortune by creating wealth has been turned on and off. Northern Italy in 800, off (warlords would steal it). Northern Italy in 1100, on. Central France in 1100, off (still feudal). England in 1800, on. England in 1974, off (98% tax on investment income). United States in 1974, on. We've even had a twin study: West Germany, on; East Germany, off. In every case, the creation of wealth seems to appear and disappear like the noise of a fan as you switch on and off the prospect of keeping it.

There is some momentum involved. It probably takes at least a generation to turn people into East Germans (luckily for England). But if it were merely a fan we were studying, without all the extra baggage that comes from the controversial topic of wealth, no one would have any doubt that the fan was causing the noise.

If you suppress variations in income, whether by stealing private fortunes, as feudal rulers used to do, or by taxing them away, as some modern governments have done, the result always seems to be the same. Society as a whole ends up poorer.

If I had a choice of living in a society where I was materially much better off than I am now, but was among the poorest, or in one where I was the richest, but much worse off than I am now, I'd take the first option. If I had children, it would arguably be immoral not to. It's absolute poverty you want to avoid, not relative poverty. If, as the evidence so far implies, you have to have one or the other in your society, take relative poverty.

You need rich people in your society not so much because in spending their money they create jobs, but because of what they have to do to *get* rich. I'm not talking about the trickle-down effect here. I'm not saying that if you let Henry Ford get rich, he'll hire you as a waiter at his next party. I'm saying that he'll make you a tractor to replace your horse.

Notes

[1] Part of the reason this subject is so contentious is that some of those most vocal on the subject of wealth—university students, heirs, professors, politicians, and journalists—have the least experience creating it. (This phenomenon will be familiar to anyone who has overheard conversations about sports in a bar.)

Students are mostly still on the parental dole, and have not stopped to think about where that money comes from. Heirs will be on the parental dole for life. Professors and politicians live within socialist eddies of the economy, at one remove from the creation of wealth, and are paid a flat rate regardless of how hard they work. And journalists as part of their professional code segregate themselves from the revenue-collecting half of the businesses they work for (the ad sales department). Many of these people never come face to face with the fact that the money they receive represents wealth—wealth that, except in the case of journalists, someone else created earlier. They live in a world in which income *is* doled out by a central authority according to some abstract notion of fairness (or randomly, in the case of heirs), rather than given by other people in return for something they wanted, so it may seem to them unfair that things don't work the same in the rest of the economy.

(Some professors do create a great deal of wealth for society. But the money they're paid isn't a *quid pro quo*. It's more in the nature of an investment.)

[2] When one reads about the origins of the Fabian Society, it sounds like something cooked up by the high-minded Edwardian child-heroes of Edith Nesbit's *The Wouldbegoods*.

[3] According to a study by the Corporate Library, the median total compensation, including salary, bonus, stock grants, and the exercise of stock options, of S&P 500 CEOs in 2002 was $3.65 million. According to *Sports Illustrated*, the average NBA player's salary during the 2002-03 season was $4.54 million, and the average major league baseball player's salary at the start of the 2003 season was $2.56

million. According to the Bureau of Labor Statistics, the mean annual wage in the US in 2002 was $35,560.

[4] In the early empire the price of an ordinary adult slave seems to have been about 2,000 sestertii (e.g. Horace, *Sat.* ii.7.43). A servant girl cost 600 (Martial vi.66), while Columella (iii.3.8) says that a skilled vine-dresser was worth 8,000. A doctor, P. Decimus Eros Merula, paid 50,000 sestertii for his freedom (Dessau, *Inscriptiones* 7812). Seneca (*Ep.* xxvii.7) reports that one Calvisius Sabinus paid 100,000 sestertii apiece for slaves learned in the Greek classics. Pliny (*Hist. Nat.* vii.39) says that the highest price paid for a slave up to his time was 700,000 sestertii, for the linguist (and presumably teacher) Daphnis, but that this had since been exceeded by actors buying their own freedom.

Classical Athens saw a similar variation in prices. An ordinary laborer was worth about 125 to 150 drachmae. Xenophon (*Mem.* ii.5) mentions prices ranging from 50 to 6,000 drachmae (for the manager of a silver mine).

For more on the economics of ancient slavery see:

Jones, A. H. M., "Slavery in the Ancient World," *Economic History Review*, 2:9 (1956), 185-199, reprinted in Finley, M. I. (ed.), *Slavery in Classical Antiquity*, Heffer, 1964.

[5] Eratosthenes (276—195 BC) used shadow lengths in different cities to estimate the Earth's circumference. He was off by only about 2%.

[6] No, and Windows, respectively.

[7] One of the biggest divergences between the Daddy Model and reality is the valuation of hard work. In the Daddy Model, hard work is in itself deserving. In reality, wealth is measured by what one delivers, not how much effort it costs. If I paint someone's house, the owner shouldn't pay me extra for doing it with a toothbrush.

It will seem to someone still implicitly operating on the Daddy Model that it is unfair when someone works hard and doesn't get paid much. To help clarify the matter, get rid of everyone else and put our worker on a desert island, hunting and gathering fruit. If he's bad at it he'll work very hard and not end up with much food. Is this unfair? Who is being unfair to him?

[8] Part of the reason for the tenacity of the Daddy Model may be the dual meaning of "distribution." When economists talk about "distribution of income," they mean statistical distribution. But when you use the phrase frequently, you can't help associating it with the other sense of the word (as in e.g. "distribution of alms"), and thereby subconsciously seeing wealth as something that flows from some central tap. The word "regressive" as applied to tax rates has a similar effect, at least on me; how can anything *regressive* be good?

[9] "From the beginning of the reign Thomas Lord Roos was an assiduous courtier of the young Henry VIII and was soon to reap the rewards. In 1525 he was made a Knight of the Garter and given the Earldom of Rutland. In the thirties his support of the breach with Rome, his zeal in crushing the Pilgrimage of Grace, and his readiness to vote the death-penalty in the succession of spectacular treason trials that punctuated Henry's erratic matrimonial progress made him an obvious candidate for grants of monastic property."

Stone, Lawrence, *Family and Fortune: Studies in Aristocratic Finance in the Sixteenth and Seventeenth Centuries*, Oxford University Press, 1973, p. 166.

[10] There is archaeological evidence for large settlements earlier, but it's hard to say what was happening in them.

Hodges, Richard and David Whitehouse, *Mohammed, Charlemagne and the Origins of Europe*, Cornell University Press, 1983.

[11] William Cecil and his son Robert were each in turn the most powerful minister of the crown, and both used their position to amass fortunes among the largest of their times. Robert in particular took bribery to the point of treason. "As Secretary of State and the leading advisor to King James on foreign policy, [he] was a special recipient of favour, being offered large bribes by the Dutch not to make peace with Spain, and large bribes by Spain to make peace." (Stone, *op. cit.*, p. 17.)

[12] Though Balzac made a lot of money from writing, he was notoriously improvident and was troubled by debts all his life.

[13] A Timex will gain or lose about .5 seconds per day. The most accurate mechanical watch, the Patek Philippe 10 Day Tourbillon, is rated at -1.5 to +2 seconds. Its retail price is about $220,000.

[14] If asked to choose which was more expensive, a well-preserved 1989 Lincoln Town Car ten-passenger limousine ($5,000) or a 2004 Mercedes S600 sedan ($122,000), the average Edwardian might well guess wrong.

[15] To say anything meaningful about income trends, you have to talk about real income, or income as measured in what it can buy. But the usual way of calculating real income ignores much of the growth in wealth over time, because it depends on a consumer price index created by bolting end to end a series of numbers that are only locally accurate, and that don't include the prices of new inventions until they become so common that their prices stabilize.

So while we might think it was very much better to live in a world with antibiotics or air travel or an electric power grid than without, real income statistics calculated in the usual way will prove to us that we are only slightly richer for having these things.

Another approach would be to ask, if you were going back to the year x in a time

machine, how much would you have to spend on trade goods to make your fortune? For example, if you were going back to 1970 it would certainly be less than $500, because the processing power you can get for $500 today would have been worth at least $150 million in 1970. The function goes asymptotic fairly quickly, because for times over a hundred years or so you could get all you needed in present-day trash. In 1800 an empty plastic drink bottle with a screw top would have seemed a miracle of workmanship.

[16] Some will say this amounts to the same thing, because the rich have better opportunities for education. That's a valid point. It is still possible, to a degree, to buy your kids' way into top colleges by sending them to private schools that in effect hack the college admissions process.

According to a 2002 report by the National Center for Education Statistics, about 1.7% of American kids attend private, non-sectarian schools. At Princeton, 36% of the class of 2007 came from such schools. (Interestingly, the number at Harvard is significantly lower, about 28%.) Obviously this is a huge loophole. It does at least seem to be closing, not widening.

Perhaps the designers of admissions processes should take a lesson from the example of computer security, and instead of just assuming that their system can't be hacked, measure the degree to which it is.

∎

Spanish Translation

Great Hackers

July 2004

(This essay is derived from a talk at Oscon 2004.)

A few months ago I finished a new [book](), and in reviews I keep noticing words like "provocative'' and "controversial.'' To say nothing of "idiotic.''

I didn't mean to make the book controversial. I was trying to make it efficient. I didn't want to waste people's time telling them things they already knew. It's more efficient just to give them the diffs. But I suppose that's bound to yield an alarming book.

Edisons

There's no controversy about which idea is most controversial: the suggestion that variation in wealth might not be as big a problem as we think.

I didn't say in the book that variation in wealth was in itself a good thing. I said in some situations it might be a sign of good things. A throbbing headache is not a good thing, but it can be a sign of a good thing-- for example, that you're recovering consciousness after being hit on the head.

Variation in wealth can be a sign of variation in productivity. (In a society of one, they're identical.) And *that* is almost certainly a good thing: if your society has no variation in productivity, it's probably not because everyone is Thomas Edison. It's probably because you have no Thomas Edisons.

In a low-tech society you don't see much variation in productivity. If you have a tribe of nomads collecting sticks for a fire, how much more productive is the best stick gatherer going to be than the worst? A factor of two? Whereas when you hand people a complex tool like a computer, the variation in what they can do with it is enormous.

That's not a new idea. Fred Brooks wrote about it in 1974, and the study he quoted was published in 1968. But I think he underestimated the variation between programmers. He wrote about productivity in lines of code: the best programmers can solve a given problem in a tenth the time. But what if the

problem isn't given? In programming, as in many fields, the hard part isn't solving problems, but deciding what problems to solve. Imagination is hard to measure, but in practice it dominates the kind of productivity that's measured in lines of code.

Productivity varies in any field, but there are few in which it varies so much. The variation between programmers is so great that it becomes a difference in kind. I don't think this is something intrinsic to programming, though. In every field, technology magnifies differences in productivity. I think what's happening in programming is just that we have a lot of technological leverage. But in every field the lever is getting longer, so the variation we see is something that more and more fields will see as time goes on. And the success of companies, and countries, will depend increasingly on how they deal with it.

If variation in productivity increases with technology, then the contribution of the most productive individuals will not only be disproportionately large, but will actually grow with time. When you reach the point where 90% of a group's output is created by 1% of its members, you lose big if something (whether Viking raids, or central planning) drags their productivity down to the average.

If we want to get the most out of them, we need to understand these especially productive people. What motivates them? What do they need to do their jobs? How do you recognize them? How do you get them to come and work for you? And then of course there's the question, how do you become one?

More than Money

I know a handful of super-hackers, so I sat down and thought about what they have in common. Their defining quality is probably that they really love to program. Ordinary programmers write code to pay the bills. Great hackers think of it as something they do for fun, and which they're delighted to find people will pay them for.

Great programmers are sometimes said to be indifferent to money. This isn't quite true. It is true that all they really care about is doing interesting work. But if you make enough money, you get to work on whatever you want, and for that reason hackers *are* attracted by the idea of making really large amounts of money. But as long as they still have to show up for work every day, they care more about what they do there than how much they get paid for it.

Economically, this is a fact of the greatest importance, because it means you don't have to pay great hackers anything like what they're worth. A great programmer might be ten or a hundred times as productive as an ordinary one, but he'll consider himself lucky to get paid three times as much. As I'll explain later, this is partly because great hackers don't know how good they are. But it's also because money is not the main thing they want.

What do hackers want? Like all craftsmen, hackers like good tools. In fact, that's

an understatement. Good hackers find it unbearable to use bad tools. They'll simply refuse to work on projects with the wrong infrastructure.

At a startup I once worked for, one of the things pinned up on our bulletin board was an ad from IBM. It was a picture of an AS400, and the headline read, I think, "hackers despise it." [1]

When you decide what infrastructure to use for a project, you're not just making a technical decision. You're also making a social decision, and this may be the more important of the two. For example, if your company wants to write some software, it might seem a prudent choice to write it in Java. But when you choose a language, you're also choosing a community. The programmers you'll be able to hire to work on a Java project won't be as [smart](#) as the ones you could get to work on a project written in Python. And the quality of your hackers probably matters more than the language you choose. Though, frankly, the fact that good hackers prefer Python to Java should tell you something about the relative merits of those languages.

Business types prefer the most popular languages because they view languages as standards. They don't want to bet the company on Betamax. The thing about languages, though, is that they're not just standards. If you have to move bits over a network, by all means use TCP/IP. But a programming language isn't just a format. A programming language is a medium of expression.

I've read that Java has just overtaken Cobol as the most popular language. As a standard, you couldn't wish for more. But as a medium of expression, you could do a lot better. Of all the great programmers I can think of, I know of only one who would voluntarily program in Java. And of all the great programmers I can think of who don't work for Sun, on Java, I know of zero.

Great hackers also generally insist on using open source software. Not just because it's better, but because it gives them more control. Good hackers insist on control. This is part of what makes them good hackers: when something's broken, they need to fix it. You want them to feel this way about the software they're writing for you. You shouldn't be surprised when they feel the same way about the operating system.

A couple years ago a venture capitalist friend told me about a new startup he was involved with. It sounded promising. But the next time I talked to him, he said they'd decided to build their software on Windows NT, and had just hired a very experienced NT developer to be their chief technical officer. When I heard this, I thought, these guys are doomed. One, the CTO couldn't be a first rate hacker, because to become an eminent NT developer he would have had to use NT voluntarily, multiple times, and I couldn't imagine a great hacker doing that; and two, even if he was good, he'd have a hard time hiring anyone good to work for him if the project had to be built on NT. [2]

The Final Frontier

After software, the most important tool to a hacker is probably his office. Big companies think the function of office space is to express rank. But hackers use their offices for more than that: they use their office as a place to think in. And if you're a technology company, their thoughts are your product. So making hackers work in a noisy, distracting environment is like having a paint factory where the air is full of soot.

The cartoon strip Dilbert has a lot to say about cubicles, and with good reason. All the hackers I know despise them. The mere prospect of being interrupted is enough to prevent hackers from working on hard problems. If you want to get real work done in an office with cubicles, you have two options: work at home, or come in early or late or on a weekend, when no one else is there. Don't companies realize this is a sign that something is broken? An office environment is supposed to be something that *helps* you work, not something you work despite.

Companies like Cisco are proud that everyone there has a cubicle, even the CEO. But they're not so advanced as they think; obviously they still view office space as a badge of rank. Note too that Cisco is famous for doing very little product development in house. They get new technology by buying the startups that created it-- where presumably the hackers did have somewhere quiet to work.

One big company that understands what hackers need is Microsoft. I once saw a recruiting ad for Microsoft with a big picture of a door. Work for us, the premise was, and we'll give you a place to work where you can actually get work done. And you know, Microsoft is remarkable among big companies in that they are able to develop software in house. Not well, perhaps, but well enough.

If companies want hackers to be productive, they should look at what they do at home. At home, hackers can arrange things themselves so they can get the most done. And when they work at home, hackers don't work in noisy, open spaces; they work in rooms with doors. They work in cosy, neighborhoody places with people around and somewhere to walk when they need to mull something over, instead of in glass boxes set in acres of parking lots. They have a sofa they can take a nap on when they feel tired, instead of sitting in a coma at their desk, pretending to work. There's no crew of people with vacuum cleaners that roars through every evening during the prime hacking hours. There are no meetings or, God forbid, corporate retreats or team-building exercises. And when you look at what they're doing on that computer, you'll find it reinforces what I said earlier about tools. They may have to use Java and Windows at work, but at home, where they can choose for themselves, you're more likely to find them using Perl and Linux.

Indeed, these statistics about Cobol or Java being the most popular language can be misleading. What we ought to look at, if we want to know what tools are best, is what hackers choose when they can choose freely-- that is, in projects of their own. When you ask that question, you find that open source operating systems already have a dominant market share, and the number one language is probably

Perl.

Interesting

Along with good tools, hackers want interesting projects. What makes a project interesting? Well, obviously overtly sexy applications like stealth planes or special effects software would be interesting to work on. But any application can be interesting if it poses novel technical challenges. So it's hard to predict which problems hackers will like, because some become interesting only when the people working on them discover a new kind of solution. Before ITA (who wrote the software inside Orbitz), the people working on airline fare searches probably thought it was one of the most boring applications imaginable. But ITA made it interesting by redefining the problem in a more ambitious way.

I think the same thing happened at Google. When Google was founded, the conventional wisdom among the so-called portals was that search was boring and unimportant. But the guys at Google didn't think search was boring, and that's why they do it so well.

This is an area where managers can make a difference. Like a parent saying to a child, I bet you can't clean up your whole room in ten minutes, a good manager can sometimes redefine a problem as a more interesting one. Steve Jobs seems to be particularly good at this, in part simply by having high standards. There were a lot of small, inexpensive computers before the Mac. He redefined the problem as: make one that's beautiful. And that probably drove the developers harder than any carrot or stick could.

They certainly delivered. When the Mac first appeared, you didn't even have to turn it on to know it would be good; you could tell from the case. A few weeks ago I was walking along the street in Cambridge, and in someone's trash I saw what appeared to be a Mac carrying case. I looked inside, and there was a Mac SE. I carried it home and plugged it in, and it booted. The happy Macintosh face, and then the finder. My God, it was so simple. It was just like ... Google.

Hackers like to work for people with high standards. But it's not enough just to be exacting. You have to insist on the right things. Which usually means that you have to be a hacker yourself. I've seen occasional articles about how to manage programmers. Really there should be two articles: one about what to do if you are yourself a programmer, and one about what to do if you're not. And the second could probably be condensed into two words: give up.

The problem is not so much the day to day management. Really good hackers are practically self-managing. The problem is, if you're not a hacker, you can't tell who the good hackers are. A similar problem explains why American cars are so ugly. I call it the *design paradox.* You might think that you could make your products beautiful just by hiring a great designer to design them. But if you yourself don't have good taste, how are you going to recognize a good designer? By definition you can't tell from his portfolio. And you can't go by the awards he's won or the

jobs he's had, because in design, as in most fields, those tend to be driven by fashion and schmoozing, with actual ability a distant third. There's no way around it: you can't manage a process intended to produce beautiful things without knowing what beautiful is. American cars are ugly because American car companies are run by people with bad taste.

Many people in this country think of taste as something elusive, or even frivolous. It is neither. To drive design, a manager must be the most demanding user of a company's products. And if you have really good taste, you can, as Steve Jobs does, make satisfying you the kind of problem that good people like to work on.

Nasty Little Problems

It's pretty easy to say what kinds of problems are not interesting: those where instead of solving a few big, clear, problems, you have to solve a lot of nasty little ones. One of the worst kinds of projects is writing an interface to a piece of software that's full of bugs. Another is when you have to customize something for an individual client's complex and ill-defined needs. To hackers these kinds of projects are the death of a thousand cuts.

The distinguishing feature of nasty little problems is that you don't learn anything from them. Writing a compiler is interesting because it teaches you what a compiler is. But writing an interface to a buggy piece of software doesn't teach you anything, because the bugs are random. [3] So it's not just fastidiousness that makes good hackers avoid nasty little problems. It's more a question of self-preservation. Working on nasty little problems makes you stupid. Good hackers avoid it for the same reason models avoid cheeseburgers.

Of course some problems inherently have this character. And because of supply and demand, they pay especially well. So a company that found a way to get great hackers to work on tedious problems would be very successful. How would you do it?

One place this happens is in startups. At our startup we had Robert Morris working as a system administrator. That's like having the Rolling Stones play at a bar mitzvah. You can't hire that kind of talent. But people will do any amount of drudgery for companies of which they're the founders. [4]

Bigger companies solve the problem by partitioning the company. They get smart people to work for them by establishing a separate R&D department where employees don't have to work directly on customers' nasty little problems. [5] In this model, the research department functions like a mine. They produce new ideas; maybe the rest of the company will be able to use them.

You may not have to go to this extreme. Bottom-up programming suggests another way to partition the company: have the smart people work as toolmakers. If your company makes software to do x, have one group that builds tools for writing software of that type, and another that uses these tools to write the

applications. This way you might be able to get smart people to write 99% of your code, but still keep them almost as insulated from users as they would be in a traditional research department. The toolmakers would have users, but they'd only be the company's own developers. [6]

If Microsoft used this approach, their software wouldn't be so full of security holes, because the less smart people writing the actual applications wouldn't be doing low-level stuff like allocating memory. Instead of writing Word directly in C, they'd be plugging together big Lego blocks of Word-language. (Duplo, I believe, is the technical term.)

Clumping

Along with interesting problems, what good hackers like is other good hackers. Great hackers tend to clump together-- sometimes spectacularly so, as at Xerox Parc. So you won't attract good hackers in linear proportion to how good an environment you create for them. The tendency to clump means it's more like the square of the environment. So it's winner take all. At any given time, there are only about ten or twenty places where hackers most want to work, and if you aren't one of them, you won't just have fewer great hackers, you'll have zero.

Having great hackers is not, by itself, enough to make a company successful. It works well for Google and ITA, which are two of the hot spots right now, but it didn't help Thinking Machines or Xerox. Sun had a good run for a while, but their business model is a down elevator. In that situation, even the best hackers can't save you.

I think, though, that all other things being equal, a company that can attract great hackers will have a huge advantage. There are people who would disagree with this. When we were making the rounds of venture capital firms in the 1990s, several told us that software companies didn't win by writing great software, but through brand, and dominating channels, and doing the right deals.

They really seemed to believe this, and I think I know why. I think what a lot of VCs are looking for, at least unconsciously, is the next Microsoft. And of course if Microsoft is your model, you shouldn't be looking for companies that hope to win by writing great software. But VCs are mistaken to look for the next Microsoft, because no startup can be the next Microsoft unless some other company is prepared to bend over at just the right moment and be the next IBM.

It's a mistake to use Microsoft as a model, because their whole culture derives from that one lucky break. Microsoft is a bad data point. If you throw them out, you find that good products do tend to win in the market. What VCs should be looking for is the next Apple, or the next Google.

I think Bill Gates knows this. What worries him about Google is not the power of their brand, but the fact that they have better hackers. [7]

Recognition

So who are the great hackers? How do you know when you meet one? That turns out to be very hard. Even hackers can't tell. I'm pretty sure now that my friend Trevor Blackwell is a great hacker. You may have read on Slashdot how he made his [own Segway](). The remarkable thing about this project was that he wrote all the software in one day (in Python, incidentally).

For Trevor, that's par for the course. But when I first met him, I thought he was a complete idiot. He was standing in Robert Morris's office babbling at him about something or other, and I remember standing behind him making frantic gestures at Robert to shoo this nut out of his office so we could go to lunch. Robert says he misjudged Trevor at first too. Apparently when Robert first met him, Trevor had just begun a new scheme that involved writing down everything about every aspect of his life on a stack of index cards, which he carried with him everywhere. He'd also just arrived from Canada, and had a strong Canadian accent and a mullet.

The problem is compounded by the fact that hackers, despite their reputation for social obliviousness, sometimes put a good deal of effort into seeming smart. When I was in grad school I used to hang around the MIT AI Lab occasionally. It was kind of intimidating at first. Everyone there spoke so fast. But after a while I learned the trick of speaking fast. You don't have to think any faster; just use twice as many words to say everything.

With this amount of noise in the signal, it's hard to tell good hackers when you meet them. I can't tell, even now. You also can't tell from their resumes. It seems like the only way to judge a hacker is to work with him on something.

And this is the reason that high-tech areas only happen around universities. The active ingredient here is not so much the professors as the students. Startups grow up around universities because universities bring together promising young people and make them work on the same projects. The smart ones learn who the other smart ones are, and together they cook up new projects of their own.

Because you can't tell a great hacker except by working with him, hackers themselves can't tell how good they are. This is true to a degree in most fields. I've found that people who are great at something are not so much convinced of their own greatness as mystified at why everyone else seems so incompetent.

But it's particularly hard for hackers to know how good they are, because it's hard to compare their work. This is easier in most other fields. In the hundred meters, you know in 10 seconds who's fastest. Even in math there seems to be a general consensus about which problems are hard to solve, and what constitutes a good solution. But hacking is like writing. Who can say which of two novels is better? Certainly not the authors.

With hackers, at least, other hackers can tell. That's because, unlike novelists,

hackers collaborate on projects. When you get to hit a few difficult problems over the net at someone, you learn pretty quickly how hard they hit them back. But hackers can't watch themselves at work. So if you ask a great hacker how good he is, he's almost certain to reply, I don't know. He's not just being modest. He really doesn't know.

And none of us know, except about people we've actually worked with. Which puts us in a weird situation: we don't know who our heroes should be. The hackers who become famous tend to become famous by random accidents of PR. Occasionally I need to give an example of a great hacker, and I never know who to use. The first names that come to mind always tend to be people I know personally, but it seems lame to use them. So, I think, maybe I should say Richard Stallman, or Linus Torvalds, or Alan Kay, or someone famous like that. But I have no idea if these guys are great hackers. I've never worked with them on anything.

If there is a Michael Jordan of hacking, no one knows, including him.

Cultivation

Finally, the question the hackers have all been wondering about: how do you become a great hacker? I don't know if it's possible to make yourself into one. But it's certainly possible to do things that make you stupid, and if you can make yourself stupid, you can probably make yourself smart too.

The key to being a good hacker may be to work on what you like. When I think about the great hackers I know, one thing they have in common is the extreme difficulty of making them work on anything they don't want to. I don't know if this is cause or effect; it may be both.

To do something well you have to love it. So to the extent you can preserve hacking as something you love, you're likely to do it well. Try to keep the sense of wonder you had about programming at age 14. If you're worried that your current job is rotting your brain, it probably is.

The best hackers tend to be smart, of course, but that's true in a lot of fields. Is there some quality that's unique to hackers? I asked some friends, and the number one thing they mentioned was curiosity. I'd always supposed that all smart people were curious-- that curiosity was simply the first derivative of knowledge. But apparently hackers are particularly curious, especially about how things work. That makes sense, because programs are in effect giant descriptions of how things work.

Several friends mentioned hackers' ability to concentrate-- their ability, as one put it, to "tune out everything outside their own heads." I've certainly noticed this. And I've heard several hackers say that after drinking even half a beer they can't program at all. So maybe hacking does require some special ability to focus. Perhaps great hackers can load a large amount of context into their head, so that when they look at a line of code, they see not just that line but the whole program

around it. John McPhee wrote that Bill Bradley's success as a basketball player was due partly to his extraordinary peripheral vision. "Perfect'' eyesight means about 47 degrees of vertical peripheral vision. Bill Bradley had 70; he could see the basket when he was looking at the floor. Maybe great hackers have some similar inborn ability. (I cheat by using a very dense language, which shrinks the court.)

This could explain the disconnect over cubicles. Maybe the people in charge of facilities, not having any concentration to shatter, have no idea that working in a cubicle feels to a hacker like having one's brain in a blender. (Whereas Bill, if the rumors of autism are true, knows all too well.)

One difference I've noticed between great hackers and smart people in general is that hackers are more politically incorrect. To the extent there is a secret handshake among good hackers, it's when they know one another well enough to express opinions that would get them stoned to death by the general public. And I can see why political incorrectness would be a useful quality in programming. Programs are very complex and, at least in the hands of good programmers, very fluid. In such situations it's helpful to have a habit of questioning assumptions.

Can you cultivate these qualities? I don't know. But you can at least not repress them. So here is my best shot at a recipe. If it is possible to make yourself into a great hacker, the way to do it may be to make the following deal with yourself: you never have to work on boring projects (unless your family will starve otherwise), and in return, you'll never allow yourself to do a half-assed job. All the great hackers I know seem to have made that deal, though perhaps none of them had any choice in the matter.

Notes

[1] In fairness, I have to say that IBM makes decent hardware. I wrote this on an IBM laptop.

[2] They did turn out to be doomed. They shut down a few months later.

[3] I think this is what people mean when they talk about the "meaning of life." On the face of it, this seems an odd idea. Life isn't an expression; how could it have meaning? But it can have a quality that feels a lot like meaning. In a project like a compiler, you have to solve a lot of problems, but the problems all fall into a pattern, as in a signal. Whereas when the problems you have to solve are random, they seem like noise.

[4] Einstein at one point worked designing refrigerators. (He had equity.)

[5] It's hard to say exactly what constitutes research in the computer world, but as

a first approximation, it's software that doesn't have users.

I don't think it's publication that makes the best hackers want to work in research departments. I think it's mainly not having to have a three hour meeting with a product manager about problems integrating the Korean version of Word 13.27 with the talking paperclip.

[6] Something similar has been happening for a long time in the construction industry. When you had a house built a couple hundred years ago, the local builders built everything in it. But increasingly what builders do is assemble components designed and manufactured by someone else. This has, like the arrival of desktop publishing, given people the freedom to experiment in disastrous ways, but it is certainly more efficient.

[7] Google is much more dangerous to Microsoft than Netscape was. Probably more dangerous than any other company has ever been. Not least because they're determined to fight. On their job listing page, they say that one of their "core values'' is "Don't be evil.'' From a company selling soybean oil or mining equipment, such a statement would merely be eccentric. But I think all of us in the computer world recognize who that is a declaration of war on.

Thanks to Jessica Livingston, Robert Morris, and Sarah Harlin for reading earlier versions of this talk.

[Audio of talk](#)

[The Python Paradox](#)

[Japanese Translation](#)

[Russian Translation](#)

[Italian Translation](#)

[Spanish Translation](#)

If you liked this, you may also like ***[Hackers & Painters](#)***.

The Python Paradox

August 2004

In a recent [talk](#) I said something that upset a lot of people: that you could get smarter programmers to work on a Python project than you could to work on a Java project.

I didn't mean by this that Java programmers are dumb. I meant that Python programmers are smart. It's a lot of work to learn a new programming language. And people don't learn Python because it will get them a job; they learn it because they genuinely like to program and aren't satisfied with the languages they already know.

Which makes them exactly the kind of programmers companies should want to hire. Hence what, for lack of a better name, I'll call the Python paradox: if a company chooses to write its software in a comparatively esoteric language, they'll be able to hire better programmers, because they'll attract only those who cared enough to learn it. And for programmers the paradox is even more pronounced: the language to learn, if you want to get a good job, is a language that people don't learn merely to get a job.

Only a few companies have been smart enough to realize this so far. But there is a kind of selection going on here too: they're exactly the companies programmers would most like to work for. Google, for example. When they advertise Java programming jobs, they also want Python experience.

A friend of mine who knows nearly all the widely used languages uses Python for most of his projects. He says the main reason is that he likes the way source code looks. That may seem a frivolous reason to choose one language over another. But it is not so frivolous as it sounds: when you program, you spend more time reading code than writing it. You push blobs of source code around the way a sculptor does blobs of clay. So a language that makes source code ugly is maddening to an exacting programmer, as clay full of lumps would be to a sculptor.

At the mention of ugly source code, people will of course think of Perl. But the superficial ugliness of Perl is not the sort I mean. Real ugliness is not harsh-looking syntax, but having to build programs out of the wrong concepts. Perl may look like a cartoon character swearing, but there are [cases](#) where it surpasses Python conceptually.

So far, anyway. Both languages are of course [moving](#) targets. But they share, along with Ruby (and Icon, and Joy, and J, and Lisp, and Smalltalk) the fact that they're created by, and used by, people who really care about programming. And those tend to be the ones who do it well.

If you liked this, you may also like ***[Hackers & Painters](#)***.

The Age of the Essay

September 2004

Remember the essays you had to write in high school? Topic sentence, introductory paragraph, supporting paragraphs, conclusion. The conclusion being, say, that Ahab in *Moby Dick* was a Christ-like figure.

Oy. So I'm going to try to give the other side of the story: what an essay really is, and how you write one. Or at least, how I write one.

Mods

The most obvious difference between real essays and the things one has to write in school is that real essays are not exclusively about English literature. Certainly schools should teach students how to write. But due to a series of historical accidents the teaching of writing has gotten mixed together with the study of literature. And so all over the country students are writing not about how a baseball team with a small budget might compete with the Yankees, or the role of color in fashion, or what constitutes a good dessert, but about symbolism in Dickens.

With the result that writing is made to seem boring and pointless. Who cares about symbolism in Dickens? Dickens himself would be more interested in an essay about color or baseball.

How did things get this way? To answer that we have to go back almost a thousand years. Around 1100, Europe at last began to catch its breath after centuries of chaos, and once they had the luxury of curiosity they rediscovered what we call "the classics." The effect was rather as if we were visited by beings from another solar system. These earlier civilizations were so much more sophisticated that for the next several centuries the main work of European scholars, in almost every field, was to assimilate what they knew.

During this period the study of ancient texts acquired great prestige. It seemed the

essence of what scholars did. As European scholarship gained momentum it became less and less important; by 1350 someone who wanted to learn about science could find better teachers than Aristotle in his own era. [1] But schools change slower than scholarship. In the 19th century the study of ancient texts was still the backbone of the curriculum.

The time was then ripe for the question: if the study of ancient texts is a valid field for scholarship, why not modern texts? The answer, of course, is that the original raison d'etre of classical scholarship was a kind of intellectual archaeology that does not need to be done in the case of contemporary authors. But for obvious reasons no one wanted to give that answer. The archaeological work being mostly done, it implied that those studying the classics were, if not wasting their time, at least working on problems of minor importance.

And so began the study of modern literature. There was a good deal of resistance at first. The first courses in English literature seem to have been offered by the newer colleges, particularly American ones. Dartmouth, the University of Vermont, Amherst, and University College, London taught English literature in the 1820s. But Harvard didn't have a professor of English literature until 1876, and Oxford not till 1885. (Oxford had a chair of Chinese before it had one of English.) [2]

What tipped the scales, at least in the US, seems to have been the idea that professors should do research as well as teach. This idea (along with the PhD, the department, and indeed the whole concept of the modern university) was imported from Germany in the late 19th century. Beginning at Johns Hopkins in 1876, the new model spread rapidly.

Writing was one of the casualties. Colleges had long taught English composition. But how do you do research on composition? The professors who taught math could be required to do original math, the professors who taught history could be required to write scholarly articles about history, but what about the professors who taught rhetoric or composition? What should they do research on? The closest thing seemed to be English literature. [3]

And so in the late 19th century the teaching of writing was inherited by English professors. This had two drawbacks: (a) an expert on literature need not himself be a good writer, any more than an art historian has to be a good painter, and (b) the subject of writing now tends to be literature, since that's what the professor is interested in.

High schools imitate universities. The seeds of our miserable high school experiences were sown in 1892, when the National Education Association "formally recommended that literature and composition be unified in the high school course." [4] The 'riting component of the 3 Rs then morphed into English, with the bizarre consequence that high school students now had to write about English literature-- to write, without even realizing it, imitations of whatever English professors had been publishing in their journals a few decades before.

It's no wonder if this seems to the student a pointless exercise, because we're now three steps removed from real work: the students are imitating English professors, who are imitating classical scholars, who are merely the inheritors of a tradition growing out of what was, 700 years ago, fascinating and urgently needed work.

No Defense

The other big difference between a real essay and the things they make you write in school is that a real essay doesn't take a position and then defend it. That principle, like the idea that we ought to be writing about literature, turns out to be another intellectual hangover of long forgotten origins.

It's often mistakenly believed that medieval universities were mostly seminaries. In fact they were more law schools. And at least in our tradition lawyers are advocates, trained to take either side of an argument and make as good a case for it as they can. Whether cause or effect, this spirit pervaded early universities. The study of rhetoric, the art of arguing persuasively, was a third of the undergraduate curriculum. [5] And after the lecture the most common form of discussion was the disputation. This is at least nominally preserved in our present-day thesis defense: most people treat the words thesis and dissertation as interchangeable, but originally, at least, a thesis was a position one took and the dissertation was the argument by which one defended it.

Defending a position may be a necessary evil in a legal dispute, but it's not the best way to get at the truth, as I think lawyers would be the first to admit. It's not just that you miss subtleties this way. The real problem is that you can't change the question.

And yet this principle is built into the very structure of the things they teach you to write in high school. The topic sentence is your thesis, chosen in advance, the supporting paragraphs the blows you strike in the conflict, and the conclusion-- uh, what is the conclusion? I was never sure about that in high school. It seemed as if we were just supposed to restate what we said in the first paragraph, but in different enough words that no one could tell. Why bother? But when you understand the origins of this sort of "essay," you can see where the conclusion comes from. It's the concluding remarks to the jury.

Good writing should be convincing, certainly, but it should be convincing because you got the right answers, not because you did a good job of arguing. When I give a draft of an essay to friends, there are two things I want to know: which parts bore them, and which seem unconvincing. The boring bits can usually be fixed by cutting. But I don't try to fix the unconvincing bits by arguing more cleverly. I need to talk the matter over.

At the very least I must have explained something badly. In that case, in the course of the conversation I'll be forced to come up a with a clearer explanation, which I can just incorporate in the essay. More often than not I have to change what I was saying as well. But the aim is never to be convincing per se. As the

reader gets smarter, convincing and true become identical, so if I can convince smart readers I must be near the truth.

The sort of writing that attempts to persuade may be a valid (or at least inevitable) form, but it's historically inaccurate to call it an essay. An essay is something else.

Trying

To understand what a real essay is, we have to reach back into history again, though this time not so far. To Michel de Montaigne, who in 1580 published a book of what he called "essais." He was doing something quite different from what lawyers do, and the difference is embodied in the name. *Essayer* is the French verb meaning "to try" and an *essai* is an attempt. An essay is something you write to try to figure something out.

Figure out what? You don't know yet. And so you can't begin with a thesis, because you don't have one, and may never have one. An essay doesn't begin with a statement, but with a question. In a real essay, you don't take a position and defend it. You notice a door that's ajar, and you open it and walk in to see what's inside.

If all you want to do is figure things out, why do you need to write anything, though? Why not just sit and think? Well, there precisely is Montaigne's great discovery. Expressing ideas helps to form them. Indeed, helps is far too weak a word. Most of what ends up in my essays I only thought of when I sat down to write them. That's why I write them.

In the things you write in school you are, in theory, merely explaining yourself to the reader. In a real essay you're writing for yourself. You're thinking out loud.

But not quite. Just as inviting people over forces you to clean up your apartment, writing something that other people will read forces you to think well. So it does matter to have an audience. The things I've written just for myself are no good. They tend to peter out. When I run into difficulties, I find I conclude with a few vague questions and then drift off to get a cup of tea.

Many published essays peter out in the same way. Particularly the sort written by the staff writers of newsmagazines. Outside writers tend to supply editorials of the defend-a-position variety, which make a beeline toward a rousing (and foreordained) conclusion. But the staff writers feel obliged to write something "balanced." Since they're writing for a popular magazine, they start with the most radioactively controversial questions, from which-- because they're writing for a popular magazine-- they then proceed to recoil in terror. Abortion, for or against? This group says one thing. That group says another. One thing is certain: the question is a complex one. (But don't get mad at us. We didn't draw any conclusions.)

The River

Questions aren't enough. An essay has to come up with answers. They don't always, of course. Sometimes you start with a promising question and get nowhere. But those you don't publish. Those are like experiments that get inconclusive results. An essay you publish ought to tell the reader something he didn't already know.

But *what* you tell him doesn't matter, so long as it's interesting. I'm sometimes accused of meandering. In defend-a-position writing that would be a flaw. There you're not concerned with truth. You already know where you're going, and you want to go straight there, blustering through obstacles, and hand-waving your way across swampy ground. But that's not what you're trying to do in an essay. An essay is supposed to be a search for truth. It would be suspicious if it didn't meander.

The Meander (aka Menderes) is a river in Turkey. As you might expect, it winds all over the place. But it doesn't do this out of frivolity. The path it has discovered is the most economical route to the sea. [6]

The river's algorithm is simple. At each step, flow down. For the essayist this translates to: flow interesting. Of all the places to go next, choose the most interesting. One can't have quite as little foresight as a river. I always know generally what I want to write about. But not the specific conclusions I want to reach; from paragraph to paragraph I let the ideas take their course.

This doesn't always work. Sometimes, like a river, one runs up against a wall. Then I do the same thing the river does: backtrack. At one point in this essay I found that after following a certain thread I ran out of ideas. I had to go back seven paragraphs and start over in another direction.

Fundamentally an essay is a train of thought-- but a cleaned-up train of thought, as dialogue is cleaned-up conversation. Real thought, like real conversation, is full of false starts. It would be exhausting to read. You need to cut and fill to emphasize the central thread, like an illustrator inking over a pencil drawing. But don't change so much that you lose the spontaneity of the original.

Err on the side of the river. An essay is not a reference work. It's not something you read looking for a specific answer, and feel cheated if you don't find it. I'd much rather read an essay that went off in an unexpected but interesting direction than one that plodded dutifully along a prescribed course.

Surprise

So what's interesting? For me, interesting means surprise. Interfaces, as Geoffrey James has said, should follow the principle of least astonishment. A button that looks like it will make a machine stop should make it stop, not speed up. Essays should do the opposite. Essays should aim for maximum surprise.

I was afraid of flying for a long time and could only travel vicariously. When friends came back from faraway places, it wasn't just out of politeness that I asked what they saw. I really wanted to know. And I found the best way to get information out of them was to ask what surprised them. How was the place different from what they expected? This is an extremely useful question. You can ask it of the most unobservant people, and it will extract information they didn't even know they were recording.

Surprises are things that you not only didn't know, but that contradict things you thought you knew. And so they're the most valuable sort of fact you can get. They're like a food that's not merely healthy, but counteracts the unhealthy effects of things you've already eaten.

How do you find surprises? Well, therein lies half the work of essay writing. (The other half is expressing yourself well.) The trick is to use yourself as a proxy for the reader. You should only write about things you've thought about a lot. And anything you come across that surprises you, who've thought about the topic a lot, will probably surprise most readers.

For example, in a recent [essay](#) I pointed out that because you can only judge computer programmers by working with them, no one knows who the best programmers are overall. I didn't realize this when I began that essay, and even now I find it kind of weird. That's what you're looking for.

So if you want to write essays, you need two ingredients: a few topics you've thought about a lot, and some ability to ferret out the unexpected.

What should you think about? My guess is that it doesn't matter-- that anything can be interesting if you get deeply enough into it. One possible exception might be things that have deliberately had all the variation sucked out of them, like working in fast food. In retrospect, was there anything interesting about working at Baskin-Robbins? Well, it was interesting how important color was to the customers. Kids a certain age would point into the case and say that they wanted yellow. Did they want French Vanilla or Lemon? They would just look at you blankly. They wanted yellow. And then there was the mystery of why the perennial favorite Pralines 'n' Cream was so appealing. (I think now it was the salt.) And the difference in the way fathers and mothers bought ice cream for their kids: the fathers like benevolent kings bestowing largesse, the mothers harried, giving in to pressure. So, yes, there does seem to be some material even in fast food.

I didn't notice those things at the time, though. At sixteen I was about as observant as a lump of rock. I can see more now in the fragments of memory I preserve of that age than I could see at the time from having it all happening live, right in front of me.

Observation

So the ability to ferret out the unexpected must not merely be an inborn one. It

must be something you can learn. How do you learn it?

To some extent it's like learning history. When you first read history, it's just a whirl of names and dates. Nothing seems to stick. But the more you learn, the more hooks you have for new facts to stick onto-- which means you accumulate knowledge at an exponential rate. Once you remember that Normans conquered England in 1066, it will catch your attention when you hear that other Normans conquered southern Italy at about the same time. Which will make you wonder about Normandy, and take note when a third book mentions that Normans were not, like most of what is now called France, tribes that flowed in as the Roman empire collapsed, but Vikings (norman = north man) who arrived four centuries later in 911. Which makes it easier to remember that Dublin was also established by Vikings in the 840s. Etc, etc squared.

Collecting surprises is a similar process. The more anomalies you've seen, the more easily you'll notice new ones. Which means, oddly enough, that as you grow older, life should become more and more surprising. When I was a kid, I used to think adults had it all figured out. I had it backwards. Kids are the ones who have it all figured out. They're just mistaken.

When it comes to surprises, the rich get richer. But (as with wealth) there may be habits of mind that will help the process along. It's good to have a habit of asking questions, especially questions beginning with Why. But not in the random way that three year olds ask why. There are an infinite number of questions. How do you find the fruitful ones?

I find it especially useful to ask why about things that seem wrong. For example, why should there be a connection between humor and misfortune? Why do we find it funny when a character, even one we like, slips on a banana peel? There's a whole essay's worth of surprises there for sure.

If you want to notice things that seem wrong, you'll find a degree of skepticism helpful. I take it as an axiom that we're only achieving 1% of what we could. This helps counteract the rule that gets beaten into our heads as children: that things are the way they are because that is how things have to be. For example, everyone I've talked to while writing this essay felt the same about English classes-- that the whole process seemed pointless. But none of us had the balls at the time to hypothesize that it was, in fact, all a mistake. We all thought there was just something we weren't getting.

I have a hunch you want to pay attention not just to things that seem wrong, but things that seem wrong in a humorous way. I'm always pleased when I see someone laugh as they read a draft of an essay. But why should I be? I'm aiming for good ideas. Why should good ideas be funny? The connection may be surprise. Surprises make us laugh, and surprises are what one wants to deliver.

I write down things that surprise me in notebooks. I never actually get around to reading them and using what I've written, but I do tend to reproduce the same

thoughts later. So the main value of notebooks may be what writing things down leaves in your head.

People trying to be cool will find themselves at a disadvantage when collecting surprises. To be surprised is to be mistaken. And the essence of cool, as any fourteen year old could tell you, is *nil admirari.* When you're mistaken, don't dwell on it; just act like nothing's wrong and maybe no one will notice.

One of the keys to coolness is to avoid situations where inexperience may make you look foolish. If you want to find surprises you should do the opposite. Study lots of different things, because some of the most interesting surprises are unexpected connections between different fields. For example, jam, bacon, pickles, and cheese, which are among the most pleasing of foods, were all originally intended as methods of preservation. And so were books and paintings.

Whatever you study, include history-- but social and economic history, not political history. History seems to me so important that it's misleading to treat it as a mere field of study. Another way to describe it is *all the data we have so far.*

Among other things, studying history gives one confidence that there are good ideas waiting to be discovered right under our noses. Swords evolved during the Bronze Age out of daggers, which (like their flint predecessors) had a hilt separate from the blade. Because swords are longer the hilts kept breaking off. But it took five hundred years before someone thought of casting hilt and blade as one piece.

Disobedience

Above all, make a habit of paying attention to things you're not supposed to, either because they're "[inappropriate](#)," or not important, or not what you're supposed to be working on. If you're curious about something, trust your instincts. Follow the threads that attract your attention. If there's something you're really interested in, you'll find they have an uncanny way of leading back to it anyway, just as the conversation of people who are especially proud of something always tends to lead back to it.

For example, I've always been fascinated by comb-overs, especially the extreme sort that make a man look as if he's wearing a beret made of his own hair. Surely this is a lowly sort of thing to be interested in-- the sort of superficial quizzing best left to teenage girls. And yet there is something underneath. The key question, I realized, is how does the comber-over not see how odd he looks? And the answer is that he got to look that way *incrementally.* What began as combing his hair a little carefully over a thin patch has gradually, over 20 years, grown into a monstrosity. Gradualness is very powerful. And that power can be used for constructive purposes too: just as you can trick yourself into looking like a freak, you can trick yourself into creating something so grand that you would never have dared to *plan* such a thing. Indeed, this is just how most good software gets created. You start by writing a stripped-down kernel (how hard can it be?) and gradually it grows into a complete operating system. Hence the next leap: could

you do the same thing in painting, or in a novel?

See what you can extract from a frivolous question? If there's one piece of advice I would give about writing essays, it would be: don't do as you're told. Don't believe what you're supposed to. Don't write the essay readers expect; one learns nothing from what one expects. And don't write the way they taught you to in school.

The most important sort of disobedience is to write essays at all. Fortunately, this sort of disobedience shows signs of becoming [rampant](). It used to be that only a tiny number of officially approved writers were allowed to write essays. Magazines published few of them, and judged them less by what they said than who wrote them; a magazine might publish a story by an unknown writer if it was good enough, but if they published an essay on x it had to be by someone who was at least forty and whose job title had x in it. Which is a problem, because there are a lot of things insiders can't say precisely because they're insiders.

The Internet is changing that. Anyone can publish an essay on the Web, and it gets judged, as any writing should, by what it says, not who wrote it. Who are you to write about x? You are whatever you wrote.

Popular magazines made the period between the spread of literacy and the arrival of TV the golden age of the short story. The Web may well make this the golden age of the essay. And that's certainly not something I realized when I started writing this.

Notes

[1] I'm thinking of Oresme (c. 1323-82). But it's hard to pick a date, because there was a sudden drop-off in scholarship just as Europeans finished assimilating classical science. The cause may have been the plague of 1347; the trend in scientific progress matches the population curve.

[2] Parker, William R. "Where Do College English Departments Come From?" *College English* 28 (1966-67), pp. 339-351. Reprinted in Gray, Donald J. (ed). *The Department of English at Indiana University Bloomington 1868-1970.* Indiana University Publications.

Daniels, Robert V. *The University of Vermont: The First Two Hundred Years.* University of Vermont, 1991.

Mueller, Friedrich M. Letter to the *Pall Mall Gazette.* 1886/87. Reprinted in Bacon, Alan (ed). *The Nineteenth-Century History of English Studies.* Ashgate, 1998.

[3] I'm compressing the story a bit. At first literature took a back seat to philology, which (a) seemed more serious and (b) was popular in Germany, where many of the leading scholars of that generation had been trained.

In some cases the writing teachers were transformed *in situ* into English professors. Francis James Child, who had been Boylston Professor of Rhetoric at Harvard since 1851, became in 1876 the university's first professor of English.

[4] Parker, *op. cit.*, p. 25.

[5] The undergraduate curriculum or *trivium* (whence "trivial") consisted of Latin grammar, rhetoric, and logic. Candidates for masters' degrees went on to study the *quadrivium* of arithmetic, geometry, music, and astronomy. Together these were the seven liberal arts.

The study of rhetoric was inherited directly from Rome, where it was considered the most important subject. It would not be far from the truth to say that education in the classical world meant training landowners' sons to speak well enough to defend their interests in political and legal disputes.

[6] Trevor Blackwell points out that this isn't strictly true, because the outside edges of curves erode faster.

Thanks to Ken Anderson, Trevor Blackwell, Sarah Harlin, Jessica Livingston, Jackie McDonough, and Robert Morris for reading drafts of this.

[Russian Translation](#)

[Spanish Translation](#)

[Japanese Translation](#)

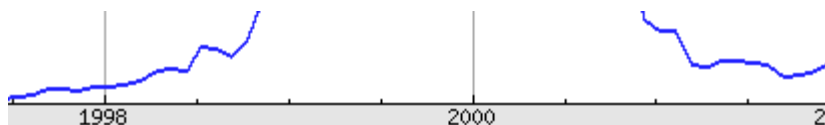[Hungarian Translation](#)

■

[Traditional Chinese Translation](#)

If you liked this, you may also like ***Hackers & Painters***.

What the Bubble Got Right

September 2004

(This essay is derived from an invited talk at ICFP 2004.)

I had a front row seat for the Internet Bubble, because I worked at Yahoo during 1998 and 1999. One day, when the stock was trading around $200, I sat down and calculated what I thought the price should be. The answer I got was $12. I went to the next cubicle and told my friend Trevor. "Twelve!" he said. He tried to sound indignant, but he didn't quite manage it. He knew as well as I did that our valuation was crazy.

Yahoo was a special case. It was not just our price to earnings ratio that was bogus. Half our earnings were too. Not in the Enron way, of course. The finance guys seemed scrupulous about reporting earnings. What made our earnings bogus was that Yahoo was, in effect, the center of a Ponzi scheme. Investors looked at Yahoo's earnings and said to themselves, here is proof that Internet companies can make money. So they invested in new startups that promised to be the next Yahoo. And as soon as these startups got the money, what did they do with it? Buy millions of dollars worth of advertising on Yahoo to promote their brand. Result: a capital investment in a startup this quarter shows up as Yahoo earnings next quarter—stimulating another round of investments in startups.

As in a Ponzi scheme, what seemed to be the returns of this system were simply the latest round of investments in it. What made it not a Ponzi scheme was that it was unintentional. At least, I think it was. The venture capital business is pretty incestuous, and there were presumably people in a position, if not to create this situation, to realize what was happening and to milk it.

A year later the game was up. Starting in January 2000, Yahoo's stock price began to crash, ultimately losing 95% of its value.

Notice, though, that even with all the fat trimmed off its market cap, Yahoo was still worth a lot. Even at the morning-after valuations of March and April 2001, the people at Yahoo had managed to create a company worth about $8 billion in just

six years.

The fact is, despite all the nonsense we heard during the Bubble about the "new economy," there was a core of truth. You need that to get a really big bubble: you need to have something solid at the center, so that even smart people are sucked in. (Isaac Newton and Jonathan Swift both lost money in the South Sea Bubble of 1720.)

Now the pendulum has swung the other way. Now anything that became fashionable during the Bubble is ipso facto unfashionable. But that's a mistake—an even bigger mistake than believing what everyone was saying in 1999. Over the long term, what the Bubble got right will be more important than what it got wrong.

1. Retail VC

After the excesses of the Bubble, it's now considered dubious to take companies public before they have earnings. But there is nothing intrinsically wrong with that idea. Taking a company public at an early stage is simply retail VC: instead of going to venture capital firms for the last round of funding, you go to the public markets.

By the end of the Bubble, companies going public with no earnings were being derided as "concept stocks," as if it were inherently stupid to invest in them. But investing in concepts isn't stupid; it's what VCs do, and the best of them are far from stupid.

The stock of a company that doesn't yet have earnings is worth *something.* It may take a while for the market to learn how to value such companies, just as it had to learn to value common stocks in the early 20th century. But markets are good at solving that kind of problem. I wouldn't be surprised if the market ultimately did a better job than VCs do now.

Going public early will not be the right plan for every company. And it can of course be disruptive—by distracting the management, or by making the early employees suddenly rich. But just as the market will learn how to value startups, startups will learn how to minimize the damage of going public.

2. The Internet

The Internet genuinely is a big deal. That was one reason even smart people were fooled by the Bubble. Obviously it was going to have a huge effect. Enough of an effect to triple the value of Nasdaq companies in two years? No, as it turned out. But it was hard to say for certain at the time. [1]

The same thing happened during the Mississippi and South Sea Bubbles. What drove them was the invention of organized public finance (the South Sea Company, despite its name, was really a competitor of the Bank of England). And that did

turn out to be a big deal, in the long run.

Recognizing an important trend turns out to be easier than figuring out how to profit from it. The mistake investors always seem to make is to take the trend too literally. Since the Internet was the big new thing, investors supposed that the more Internettish the company, the better. Hence such parodies as Pets.Com.

In fact most of the money to be made from big trends is made indirectly. It was not the railroads themselves that made the most money during the railroad boom, but the companies on either side, like Carnegie's steelworks, which made the rails, and Standard Oil, which used railroads to get oil to the East Coast, where it could be shipped to Europe.

I think the Internet will have great effects, and that what we've seen so far is nothing compared to what's coming. But most of the winners will only indirectly be Internet companies; for every Google there will be ten JetBlues.

3. Choices

Why will the Internet have great effects? The general argument is that new forms of communication always do. They happen rarely (till industrial times there were just speech, writing, and printing), but when they do, they always cause a big splash.

The specific argument, or one of them, is the Internet gives us more choices. In the "old" economy, the high cost of presenting information to people meant they had only a narrow range of options to choose from. The tiny, expensive pipeline to consumers was tellingly named "the channel." Control the channel and you could feed them what you wanted, on your terms. And it was not just big corporations that depended on this principle. So, in their way, did labor unions, the traditional news media, and the art and literary establishments. Winning depended not on doing good work, but on gaining control of some bottleneck.

There are signs that this is changing. Google has over 82 million unique users a month and annual revenues of about three billion dollars. [2] And yet have you ever seen a Google ad? Something is going on here.

Admittedly, Google is an extreme case. It's very easy for people to switch to a new search engine. It costs little effort and no money to try a new one, and it's easy to see if the results are better. And so Google doesn't *have* to advertise. In a business like theirs, being the best is enough.

The exciting thing about the Internet is that it's shifting everything in that direction. The hard part, if you want to win by making the best stuff, is the beginning. Eventually everyone will learn by word of mouth that you're the best, but how do you survive to that point? And it is in this crucial stage that the Internet has the most effect. First, the Internet lets anyone find you at almost zero cost. Second, it dramatically speeds up the rate at which reputation spreads by

word of mouth. Together these mean that in many fields the rule will be: Build it, and they will come. Make something great and put it online. That is a big change from the recipe for winning in the past century.

4. Youth

The aspect of the Internet Bubble that the press seemed most taken with was the youth of some of the startup founders. This too is a trend that will last. There is a huge standard deviation among 26 year olds. Some are fit only for entry level jobs, but others are ready to rule the world if they can find someone to handle the paperwork for them.

A 26 year old may not be very good at managing people or dealing with the SEC. Those require experience. But those are also commodities, which can be handed off to some lieutenant. The most important quality in a CEO is his vision for the company's future. What will they build next? And in that department, there are 26 year olds who can compete with anyone.

In 1970 a company president meant someone in his fifties, at least. If he had technologists working for him, they were treated like a racing stable: prized, but not powerful. But as technology has grown more important, the power of nerds has grown to reflect it. Now it's not enough for a CEO to have someone smart he can ask about technical matters. Increasingly, he has to be that person himself.

As always, business has clung to old forms. VCs still seem to want to install a legitimate-looking talking head as the CEO. But increasingly the founders of the company are the real powers, and the grey-headed man installed by the VCs more like a music group's manager than a general.

5. Informality

In New York, the Bubble had dramatic consequences: suits went out of fashion. They made one seem old. So in 1998 powerful New York types were suddenly wearing open-necked shirts and khakis and oval wire-rimmed glasses, just like guys in Santa Clara.

The pendulum has swung back a bit, driven in part by a panicked reaction by the clothing industry. But I'm betting on the open-necked shirts. And this is not as frivolous a question as it might seem. Clothes are important, as all nerds can sense, though they may not realize it consciously.

If you're a nerd, you can understand how important clothes are by asking yourself how you'd feel about a company that made you wear a suit and tie to work. The idea sounds horrible, doesn't it? In fact, horrible far out of proportion to the mere discomfort of wearing such clothes. A company that made programmers wear suits would have something deeply wrong with it.

And what would be wrong would be that how one presented oneself counted more

than the quality of one's ideas. *That's* the problem with formality. Dressing up is not so much bad in itself. The problem is the receptor it binds to: dressing up is inevitably a substitute for good ideas. It is no coincidence that technically inept business types are known as "suits."

Nerds don't just happen to dress informally. They do it too consistently. Consciously or not, they dress informally as a prophylactic measure against stupidity.

6. Nerds

Clothing is only the most visible battleground in the war against formality. Nerds tend to eschew formality of any sort. They're not impressed by one's job title, for example, or any of the other appurtenances of authority.

Indeed, that's practically the definition of a nerd. I found myself talking recently to someone from Hollywood who was planning a show about nerds. I thought it would be useful if I explained what a nerd was. What I came up with was: someone who doesn't expend any effort on marketing himself.

A nerd, in other words, is someone who concentrates on substance. So what's the connection between nerds and technology? Roughly that you can't fool mother nature. In technical matters, you have to get the right answers. If your software miscalculates the path of a space probe, you can't finesse your way out of trouble by saying that your code is patriotic, or avant-garde, or any of the other dodges people use in nontechnical fields.

And as technology becomes increasingly important in the economy, nerd culture is rising with it. Nerds are already a lot cooler than they were when I was a kid. When I was in college in the mid-1980s, "nerd" was still an insult. People who majored in computer science generally tried to conceal it. Now women ask me where they can meet nerds. (The answer that springs to mind is "Usenix," but that would be like drinking from a firehose.)

I have no illusions about why nerd culture is becoming more accepted. It's not because people are realizing that substance is more important than marketing. It's because the nerds are getting rich. But that is not going to change.

7. Options

What makes the nerds rich, usually, is stock options. Now there are moves afoot to make it harder for companies to grant options. To the extent there's some genuine accounting abuse going on, by all means correct it. But don't kill the golden goose. Equity is the fuel that drives technical innovation.

Options are a good idea because (a) they're fair, and (b) they work. Someone who goes to work for a company is (one hopes) adding to its value, and it's only fair to give them a share of it. And as a purely practical measure, people work a *lot*

harder when they have options. I've seen that first hand.

The fact that a few crooks during the Bubble robbed their companies by granting themselves options doesn't mean options are a bad idea. During the railroad boom, some executives enriched themselves by selling watered stock—by issuing more shares than they said were outstanding. But that doesn't make common stock a bad idea. Crooks just use whatever means are available.

If there is a problem with options, it's that they reward slightly the wrong thing. Not surprisingly, people do what you pay them to. If you pay them by the hour, they'll work a lot of hours. If you pay them by the volume of work done, they'll get a lot of work done (but only as you defined work). And if you pay them to raise the stock price, which is what options amount to, they'll raise the stock price.

But that's not quite what you want. What you want is to increase the actual value of the company, not its market cap. Over time the two inevitably meet, but not always as quickly as options vest. Which means options tempt employees, if only unconsciously, to "pump and dump"—to do things that will make the company *seem* valuable. I found that when I was at Yahoo, I couldn't help thinking, "how will this sound to investors?" when I should have been thinking "is this a good idea?"

So maybe the standard option deal needs to be tweaked slightly. Maybe options should be replaced with something tied more directly to earnings. It's still early days.

8. Startups

What made the options valuable, for the most part, is that they were options on the stock of startups. Startups were not of course a creation of the Bubble, but they were more visible during the Bubble than ever before.

One thing most people did learn about for the first time during the Bubble was the startup created with the intention of selling it. Originally a startup meant a small company that hoped to grow into a big one. But increasingly startups are evolving into a vehicle for developing technology on spec.

As I wrote in Hackers & Painters, employees seem to be most productive when they're paid in proportion to the wealth they generate. And the advantage of a startup—indeed, almost its raison d'etre—is that it offers something otherwise impossible to obtain: a way of *measuring* that.

In many businesses, it just makes more sense for companies to get technology by buying startups rather than developing it in house. You pay more, but there is less risk, and risk is what big companies don't want. It makes the guys developing the technology more accountable, because they only get paid if they build the winner. And you end up with better technology, created faster, because things are made in the innovative atmosphere of startups instead of the bureaucratic atmosphere of

big companies.

Our startup, Viaweb, was built to be sold. We were open with investors about that from the start. And we were careful to create something that could slot easily into a larger company. That is the pattern for the future.

9. California

The Bubble was a California phenomenon. When I showed up in Silicon Valley in 1998, I felt like an immigrant from Eastern Europe arriving in America in 1900. Everyone was so cheerful and healthy and rich. It seemed a new and improved world.

The press, ever eager to exaggerate small trends, now gives one the impression that Silicon Valley is a ghost town. Not at all. When I drive down 101 from the airport, I still feel a buzz of energy, as if there were a giant transformer nearby. Real estate is still more expensive than just about anywhere else in the country. The people still look healthy, and the weather is still fabulous. The future is there. (I say "there" because I moved back to the East Coast after Yahoo. I still wonder if this was a smart idea.)

What makes the Bay Area superior is the attitude of the people. I notice that when I come home to Boston. The first thing I see when I walk out of the airline terminal is the fat, grumpy guy in charge of the taxi line. I brace myself for rudeness: *remember, you're back on the East Coast now.*

The atmosphere varies from city to city, and fragile organisms like startups are exceedingly sensitive to such variation. If it hadn't already been hijacked as a new euphemism for liberal, the word to describe the atmosphere in the Bay Area would be "progressive." People there are trying to build the future. Boston has MIT and Harvard, but it also has a lot of truculent, unionized employees like the police who recently held the Democratic National Convention for ransom, and a lot of people trying to be Thurston Howell. Two sides of an obsolete coin.

Silicon Valley may not be the next Paris or London, but it is at least the next Chicago. For the next fifty years, that's where new wealth will come from.

10. Productivity

During the Bubble, optimistic analysts used to justify high price to earnings ratios by saying that technology was going to increase productivity dramatically. They were wrong about the specific companies, but not so wrong about the underlying principle. I think one of the big trends we'll see in the coming century is a huge increase in productivity.

Or more precisely, a huge increase in variation in productivity. Technology is a lever. It doesn't add; it multiplies. If the present range of productivity is 0 to 100, introducing a multiple of 10 increases the range from 0 to 1000.

One upshot of which is that the companies of the future may be surprisingly small. I sometimes daydream about how big you could grow a company (in revenues) without ever having more than ten people. What would happen if you outsourced everything except product development? If you tried this experiment, I think you'd be surprised at how far you could get. As Fred Brooks pointed out, small groups are intrinsically more productive, because the internal friction in a group grows as the square of its size.

Till quite recently, running a major company meant managing an army of workers. Our standards about how many employees a company should have are still influenced by old patterns. Startups are perforce small, because they can't afford to hire a lot of people. But I think it's a big mistake for companies to loosen their belts as revenues increase. The question is not whether you can afford the extra salaries. Can you afford the loss in productivity that comes from making the company bigger?

The prospect of technological leverage will of course raise the specter of unemployment. I'm surprised people still worry about this. After centuries of supposedly job-killing innovations, the number of jobs is within ten percent of the number of people who want them. This can't be a coincidence. There must be some kind of balancing mechanism.

What's New

When one looks over these trends, is there any overall theme? There does seem to be: that in the coming century, good ideas will count for more. That 26 year olds with good ideas will increasingly have an edge over 50 year olds with powerful connections. That doing good work will matter more than dressing up—or advertising, which is the same thing for companies. That people will be rewarded a bit more in proportion to the value of what they create.

If so, this is good news indeed. Good ideas always tend to win eventually. The problem is, it can take a very long time. It took decades for relativity to be accepted, and the greater part of a century to establish that central planning didn't work. So even a small increase in the rate at which good ideas win would be a momentous change—big enough, probably, to justify a name like the "new economy."

Notes

[1] Actually it's hard to say now. As Jeremy Siegel points out, if the value of a stock is its future earnings, you can't tell if it was overvalued till you see what the earnings turn out to be. While certain famous Internet stocks were almost certainly

overvalued in 1999, it is still hard to say for sure whether, e.g., the Nasdaq index was.

Siegel, Jeremy J. "What Is an Asset Price Bubble? An Operational Definition." *European Financial Management,* 9:1, 2003.

[2] The number of users comes from a 6/03 Nielsen study quoted on Google's site. (You'd think they'd have something more recent.) The revenue estimate is based on revenues of $1.35 billion for the first half of 2004, as reported in their IPO filing.

■

[The Long Tail](#)

■

[Russian Translation](#)

■

[Japanese Translation](#)

A Version 1.0

October 2004

As E. B. White said, "good writing is rewriting." I didn't realize this when I was in school. In writing, as in math and science, they only show you the finished product. You don't see all the false starts. This gives students a misleading view of how things get made.

Part of the reason it happens is that writers don't want people to see their mistakes. But I'm willing to let people see an early draft if it will show how much you have to rewrite to beat an essay into shape.

Below is the oldest version I can find of [The Age of the Essay](#) (probably the second or third day), with text that ultimately survived in red and text that later got deleted in gray. There seem to be several categories of cuts: things I got wrong, things that seem like bragging, flames, digressions, stretches of awkward prose, and unnecessary words.

I discarded more from the beginning. That's not surprising; it takes a while to hit your stride. There are more digressions at the start, because I'm not sure where I'm heading.

The amount of cutting is about average. I probably write three to four words for every one that appears in the final version of an essay.

(Before anyone gets mad at me for opinions expressed here, remember that anything you see here that's not in the final version is obviously something I chose not to publish, often because I disagree with it.)

Recently a friend said that what he liked about my essays was that they weren't written the way we'd been taught to write essays in school. You remember: topic sentence, introductory paragraph, supporting paragraphs, conclusion. It hadn't occurred to me till then that those horrible things we had to write in school were even connected to what I was doing now. But sure enough, I thought, they did call them "essays," didn't they?

Well, they're not. Those things you have to write in school are not only not essays,

they're one of the most pointless of all the pointless hoops you have to jump through in school. And I worry that they not only teach students the wrong things about writing, but put them off writing entirely.

So I'm going to give the other side of the story: what an essay really is, and how you write one. Or at least, how I write one. Students be forewarned: if you actually write the kind of essay I describe, you'll probably get bad grades. But knowing how it's really done should at least help you to understand the feeling of futility you have when you're writing the things they tell you to.

The most obvious difference between real essays and the things one has to write in school is that real essays are not exclusively about English literature. It's a fine thing for schools to teach students how to write. But for some bizarre reason (actually, a very specific bizarre reason that I'll explain in a moment), the teaching of writing has gotten mixed together with the study of literature. And so all over the country, students are writing not about how a baseball team with a small budget might compete with the Yankees, or the role of color in fashion, or what constitutes a good dessert, but about symbolism in Dickens.

With obvious results. Only a few people really care about symbolism in Dickens. The teacher doesn't. The students don't. Most of the people who've had to write PhD disserations about Dickens don't. And certainly Dickens himself would be more interested in an essay about color or baseball.

How did things get this way? To answer that we have to go back almost a thousand years. Between about 500 and 1000, life was not very good in Europe. The term "dark ages" is presently out of fashion as too judgemental (the period wasn't dark; it was just *different*), but if this label didn't already exist, it would seem an inspired metaphor. What little original thought there was took place in lulls between constant wars and had something of the character of the thoughts of parents with a new baby. The most amusing thing written during this period, Liudprand of Cremona's Embassy to Constantinople, is, I suspect, mostly inadvertantly so.

Around 1000 Europe began to catch its breath. And once they had the luxury of curiosity, one of the first things they discovered was what we call "the classics." Imagine if we were visited by aliens. If they could even get here they'd presumably know a few things we don't. Immediately Alien Studies would become the most dynamic field of scholarship: instead of painstakingly discovering things for ourselves, we could simply suck up everything they'd discovered. So it was in Europe in 1200. When classical texts began to circulate in Europe, they contained not just new answers, but new questions. (If anyone proved a theorem in christian Europe before 1200, for example, there is no record of it.)

For a couple centuries, some of the most important work being done was intellectual archaelogy. Those were also the centuries during which schools were first established. And since reading ancient texts was the essence of what scholars did then, it became the basis of the curriculum.

By 1700, someone who wanted to learn about physics didn't need to start by mastering Greek in order to read Aristotle. But schools change slower than scholarship: the study of ancient texts had such prestige that it remained the backbone of education until the late 19th century. By then it was merely a tradition. It did serve some purposes: reading a foreign language was difficult, and thus taught discipline, or at least, kept students busy; it introduced students to cultures quite different from their own; and its very uselessness made it function (like white gloves) as a social bulwark. But it certainly wasn't true, and hadn't been true for centuries, that students were serving apprenticeships in the hottest area of scholarship.

Classical scholarship had also changed. In the early era, philology actually mattered. The texts that filtered into Europe were all corrupted to some degree by the errors of translators and copyists. Scholars had to figure out what Aristotle said before they could figure out what he meant. But by the modern era such questions were answered as well as they were ever going to be. And so the study of ancient texts became less about ancientness and more about texts.

The time was then ripe for the question: if the study of ancient texts is a valid field for scholarship, why not modern texts? The answer, of course, is that the raison d'etre of classical scholarship was a kind of intellectual archaelogy that does not need to be done in the case of contemporary authors. But for obvious reasons no one wanted to give that answer. The archaeological work being mostly done, it implied that the people studying the classics were, if not wasting their time, at least working on problems of minor importance.

And so began the study of modern literature. There was some initial resistance, but it didn't last long. The limiting reagent in the growth of university departments is what parents will let undergraduates study. If parents will let their children major in x, the rest follows straightforwardly. There will be jobs teaching x, and professors to fill them. The professors will establish scholarly journals and publish one another's papers. Universities with x departments will subscribe to the journals. Graduate students who want jobs as professors of x will write dissertations about it. It may take a good long while for the more prestigious universities to cave in and establish departments in cheesier xes, but at the other end of the scale there are so many universities competing to attract students that the mere establishment of a discipline requires little more than the desire to do it.

High schools imitate universities. And so once university English departments were established in the late nineteenth century, the 'riting component of the 3 Rs was morphed into English. With the bizarre consequence that high school students now had to write about English literature-- to write, without even realizing it, imitations of whatever English professors had been publishing in their journals a few decades before. It's no wonder if this seems to the student a pointless exercise, because we're now three steps removed from real work: the students are imitating English professors, who are imitating classical scholars, who are merely the inheritors of a tradition growing out of what was, 700 years ago, fascinating and urgently needed work.

Perhaps high schools should drop English and just teach writing. The valuable part of English classes is learning to write, and that could be taught better by itself. Students learn better when they're interested in what they're doing, and it's hard to imagine a topic less interesting than symbolism in Dickens. Most of the people who write about that sort of thing professionally are not really interested in it. (Though indeed, it's been a while since they were writing about symbolism; now they're writing about gender.)

I have no illusions about how eagerly this suggestion will be adopted. Public schools probably couldn't stop teaching English even if they wanted to; they're probably required to by law. But here's a related suggestion that goes with the grain instead of against it: that universities establish a writing major. Many of the students who now major in English would major in writing if they could, and most would be better off.

It will be argued that it is a good thing for students to be exposed to their literary heritage. Certainly. But is that more important than that they learn to write well? And are English classes even the place to do it? After all, the average public high school student gets zero exposure to his artistic heritage. No disaster results. The people who are interested in art learn about it for themselves, and those who aren't don't. I find that American adults are no better or worse informed about literature than art, despite the fact that they spent years studying literature in high school and no time at all studying art. Which presumably means that what they're taught in school is rounding error compared to what they pick up on their own.

Indeed, English classes may even be harmful. In my case they were effectively aversion therapy. Want to make someone dislike a book? Force him to read it and write an essay about it. And make the topic so intellectually bogus that you could not, if asked, explain why one ought to write about it. I love to read more than anything, but by the end of high school I never read the books we were assigned. I was so disgusted with what we were doing that it became a point of honor with me to write nonsense at least as good at the other students' without having more than glanced over the book to learn the names of the characters and a few random events in it.

I hoped this might be fixed in college, but I found the same problem there. It was not the teachers. It was English. We were supposed to read novels and write essays about them. About what, and why? That no one seemed to be able to explain. Eventually by trial and error I found that what the teacher wanted us to do was pretend that the story had really taken place, and to analyze based on what the characters said and did (the subtler clues, the better) what their motives must have been. One got extra credit for motives having to do with class, as I suspect one must now for those involving gender and sexuality. I learned how to churn out such stuff well enough to get an A, but I never took another English class.

And the books we did these disgusting things to, like those we mishandled in high school, I find still have black marks against them in my mind. The one saving

grace was that English courses tend to favor pompous, dull writers like Henry James, who deserve black marks against their names anyway. One of the principles the IRS uses in deciding whether to allow deductions is that, if something is fun, it isn't work. Fields that are intellectually unsure of themselves rely on a similar principle. Reading P.G. Wodehouse or Evelyn Waugh or Raymond Chandler is too obviously pleasing to seem like serious work, as reading Shakespeare would have been before English evolved enough to make it an effort to understand him. [sh] And so good writers (just you wait and see who's still in print in 300 years) are less likely to have readers turned against them by clumsy, self-appointed tour guides.

The other big difference between a real essay and the things they make you write in school is that a real essay doesn't take a position and then defend it. That principle, like the idea that we ought to be writing about literature, turns out to be another intellectual hangover of long forgotten origins. It's often mistakenly believed that medieval universities were mostly seminaries. In fact they were more law schools. And at least in our tradition lawyers are advocates: they are trained to be able to take either side of an argument and make as good a case for it as they can.

Whether or not this is a good idea (in the case of prosecutors, it probably isn't), it tended to pervade the atmosphere of early universities. After the lecture the most common form of discussion was the disputation. This idea is at least nominally preserved in our present-day thesis defense-- indeed, in the very word thesis. Most people treat the words thesis and dissertation as interchangeable, but originally, at least, a thesis was a position one took and the dissertation was the argument by which one defended it.

I'm not complaining that we blur these two words together. As far as I'm concerned, the sooner we lose the original sense of the word thesis, the better. For many, perhaps most, graduate students, it is stuffing a square peg into a round hole to try to recast one's work as a single thesis. And as for the disputation, that seems clearly a net lose. Arguing two sides of a case may be a necessary evil in a legal dispute, but it's not the best way to get at the truth, as I think lawyers would be the first to admit.

And yet this principle is built into the very structure of the essays they teach you to write in high school. The topic sentence is your thesis, chosen in advance, the supporting paragraphs the blows you strike in the conflict, and the conclusion--- uh, what it the conclusion? I was never sure about that in high school. If your thesis was well expressed, what need was there to restate it? In theory it seemed that the conclusion of a really good essay ought not to need to say any more than QED. But when you understand the origins of this sort of "essay", you can see where the conclusion comes from. It's the concluding remarks to the jury.

What other alternative is there? To answer that we have to reach back into history again, though this time not so far. To Michel de Montaigne, inventor of the essay. He was doing something quite different from what a lawyer does, and the

difference is embodied in the name. Essayer is the French verb meaning "to try" (the cousin of our word assay), and an "essai" is an effort. An essay is something you write in order to figure something out.

Figure out what? You don't know yet. And so you can't begin with a thesis, because you don't have one, and may never have one. An essay doesn't begin with a statement, but with a question. In a real essay, you don't take a position and defend it. You see a door that's ajar, and you open it and walk in to see what's inside.

If all you want to do is figure things out, why do you need to write anything, though? Why not just sit and think? Well, there precisely is Montaigne's great discovery. Expressing ideas helps to form them. Indeed, helps is far too weak a word. 90% of what ends up in my essays was stuff I only thought of when I sat down to write them. That's why I write them.

So there's another difference between essays and the things you have to write in school. In school you are, in theory, explaining yourself to someone else. In the best case---if you're really organized---you're just writing it *down.* In a real essay you're writing for yourself. You're thinking out loud.

But not quite. Just as inviting people over forces you to clean up your apartment, writing something that you know other people will read forces you to think well. So it does matter to have an audience. The things I've written just for myself are no good. Indeed, they're bad in a particular way: they tend to peter out. When I run into difficulties, I notice that I tend to conclude with a few vague questions and then drift off to get a cup of tea.

This seems a common problem. It's practically the standard ending in blog entries--- with the addition of a "heh" or an emoticon, prompted by the all too accurate sense that something is missing.

And indeed, a lot of published essays peter out in this same way. Particularly the sort written by the staff writers of newsmagazines. Outside writers tend to supply editorials of the defend-a-position variety, which make a beeline toward a rousing (and foreordained) conclusion. But the staff writers feel obliged to write something more balanced, which in practice ends up meaning blurry. Since they're writing for a popular magazine, they start with the most radioactively controversial questions, from which (because they're writing for a popular magazine) they then proceed to recoil from in terror. Gay marriage, for or against? This group says one thing. That group says another. One thing is certain: the question is a complex one. (But don't get mad at us. We didn't draw any conclusions.)

Questions aren't enough. An essay has to come up with answers. They don't always, of course. Sometimes you start with a promising question and get nowhere. But those you don't publish. Those are like experiments that get inconclusive results. Something you publish ought to tell the reader something he didn't already know.

But *what* you tell him doesn't matter, so long as it's interesting. I'm sometimes accused of meandering. In defend-a-position writing that would be a flaw. There you're not concerned with truth. You already know where you're going, and you want to go straight there, blustering through obstacles, and hand-waving your way across swampy ground. But that's not what you're trying to do in an essay. An essay is supposed to be a search for truth. It would be suspicious if it didn't meander.

The Meander is a river in Asia Minor (aka Turkey). As you might expect, it winds all over the place. But does it do this out of frivolity? Quite the opposite. Like all rivers, it's rigorously following the laws of physics. The path it has discovered, winding as it is, represents the most economical route to the sea.

The river's algorithm is simple. At each step, flow down. For the essayist this translates to: flow interesting. Of all the places to go next, choose whichever seems most interesting.

I'm pushing this metaphor a bit. An essayist can't have quite as little foresight as a river. In fact what you do (or what I do) is somewhere between a river and a roman road-builder. I have a general idea of the direction I want to go in, and I choose the next topic with that in mind. This essay is about writing, so I do occasionally yank it back in that direction, but it is not all the sort of essay I thought I was going to write about writing.

Note too that hill-climbing (which is what this algorithm is called) can get you in trouble. Sometimes, just like a river, you run up against a blank wall. What I do then is just what the river does: backtrack. At one point in this essay I found that after following a certain thread I ran out of ideas. I had to go back n paragraphs and start over in another direction. For illustrative purposes I've left the abandoned branch as a footnote.

Err on the side of the river. An essay is not a reference work. It's not something you read looking for a specific answer, and feel cheated if you don't find it. I'd much rather read an essay that went off in an unexpected but interesting direction than one that plodded dutifully along a prescribed course.

So what's interesting? For me, interesting means surprise. Design, as Matz has said, should follow the principle of least surprise. A button that looks like it will make a machine stop should make it stop, not speed up. Essays should do the opposite. Essays should aim for maximum surprise.

I was afraid of flying for a long time and could only travel vicariously. When friends came back from faraway places, it wasn't just out of politeness that I asked them about their trip. I really wanted to know. And I found that the best way to get information out of them was to ask what surprised them. How was the place different from what they expected? This is an extremely useful question. You can ask it of even the most unobservant people, and it will extract information they

didn't even know they were recording.

Indeed, you can ask it in real time. Now when I go somewhere new, I make a note of what surprises me about it. Sometimes I even make a conscious effort to visualize the place beforehand, so I'll have a detailed image to diff with reality.

Surprises are facts you didn't already know. But they're more than that. They're facts that contradict things you thought you knew. And so they're the most valuable sort of fact you can get. They're like a food that's not merely healthy, but counteracts the unhealthy effects of things you've already eaten.

How do you find surprises? Well, therein lies half the work of essay writing. (The other half is expressing yourself well.) You can at least use yourself as a proxy for the reader. You should only write about things you've thought about a lot. And anything you come across that surprises you, who've thought about the topic a lot, will probably surprise most readers.

For example, in a recent essay I pointed out that because you can only judge computer programmers by working with them, no one knows in programming who the heroes should be. I certainly didn't realize this when I started writing the essay, and even now I find it kind of weird. That's what you're looking for.

So if you want to write essays, you need two ingredients: you need a few topics that you think about a lot, and you need some ability to ferret out the unexpected.

What should you think about? My guess is that it doesn't matter. Almost everything is interesting if you get deeply enough into it. The one possible exception are things like working in fast food, which have deliberately had all the variation sucked out of them. In retrospect, was there anything interesting about working in Baskin-Robbins? Well, it was interesting to notice how important color was to the customers. Kids a certain age would point into the case and say that they wanted yellow. Did they want French Vanilla or Lemon? They would just look at you blankly. They wanted yellow. And then there was the mystery of why the perennial favorite Pralines n' Cream was so appealing. I'm inclined now to think it was the salt. And the mystery of why Passion Fruit tasted so disgusting. People would order it because of the name, and were always disappointed. It should have been called In-sink-erator Fruit. And there was the difference in the way fathers and mothers bought ice cream for their kids. Fathers tended to adopt the attitude of benevolent kings bestowing largesse, and mothers that of harried bureaucrats, giving in to pressure against their better judgement. So, yes, there does seem to be material, even in fast food.

What about the other half, ferreting out the unexpected? That may require some natural ability. I've noticed for a long time that I'm pathologically observant. ....

[That was as far as I'd gotten at the time.]

Notes

[sh] In Shakespeare's own time, serious writing meant theological discourses, not the bawdy plays acted over on the other side of the river among the bear gardens and whorehouses.

The other extreme, the work that seems formidable from the moment it's created (indeed, is deliberately intended to be) is represented by Milton. Like the Aeneid, Paradise Lost is a rock imitating a butterfly that happened to get fossilized. Even Samuel Johnson seems to have balked at this, on the one hand paying Milton the compliment of an extensive biography, and on the other writing of Paradise Lost that "none who read it ever wished it longer."

Bradley's Ghost

November 2004

A lot of people are writing now about why Kerry lost. Here I want to examine a more specific question: why were the exit polls so wrong?

In Ohio, which Kerry ultimately lost 49-51, exit polls gave him a 52-48 victory. And this wasn't just random error. In every swing state they overestimated the Kerry vote. In Florida, which Bush ultimately won 52-47, exit polls predicted a dead heat.

(These are not early numbers. They're from about midnight eastern time, long after polls closed in Ohio and Florida. And yet by the next afternoon the exit poll numbers online corresponded to the returns. The only way I can imagine this happening is if those in charge of the exit polls cooked the books after seeing the actual returns. But that's another issue.)

What happened? The source of the problem may be a variant of the Bradley Effect. This term was invented after Tom Bradley, the black mayor of Los Angeles, lost an election for governor of California despite a comfortable lead in the polls. Apparently voters were afraid to say they planned to vote against him, lest their motives be (perhaps correctly) suspected.

It seems likely that something similar happened in exit polls this year. In theory, exit polls ought to be very accurate. You're not asking people what they would do. You're asking what they just did.

How can you get errors asking that? Because some people don't respond. To get a truly random sample, pollsters ask, say, every 20th person leaving the polling place who they voted for. But not everyone wants to answer. And the pollsters can't simply ignore those who won't, or their sample isn't random anymore. So what they do, apparently, is note down the age and race and sex of the person, and guess from that who they voted for.

This works so long as there is no *correlation* between who people vote for and

whether they're willing to talk about it. But this year there may have been. It may be that a significant number of those who voted for Bush didn't want to say so.

Why not? Because people in the US are more conservative than they're willing to admit. The values of the elite in this country, at least at the moment, are NPR values. The average person, as I think both Republicans and Democrats would agree, is more socially conservative. But while some openly flaunt the fact that they don't share the opinions of the elite, others feel a little nervous about it, as if they had bad table manners.

For example, according to current NPR values, you can't say anything that might be perceived as disparaging towards homosexuals. To do so is "homophobic." And yet a large number of Americans are deeply religious, and the Bible is quite explicit on the subject of homosexuality. What are they to do? I think what many do is keep their opinions, but keep them to themselves.

They know what they believe, but they also know what they're supposed to believe. And so when a stranger (for example, a pollster) asks them their opinion about something like gay marriage, they will not always say what they really think.

When the values of the elite are liberal, polls will tend to underestimate the conservativeness of ordinary voters. This seems to me the leading theory to explain why the exit polls were so far off this year. NPR values said one ought to vote for Kerry. So all the people who voted for Kerry felt virtuous for doing so, and were eager to tell pollsters they had. No one who voted for Kerry did it as an act of quiet defiance.

■

Support for a Woman President

■

Japanese Translation

If you liked this, you may also like *__Hackers & Painters__*.

It's Charisma, Stupid

November 2004, corrected June 2006

Occam's razor says we should prefer the simpler of two explanations. I begin by reminding readers of this principle because I'm about to propose a theory that will offend both liberals and conservatives. But Occam's razor means, in effect, that if you want to disagree with it, you have a hell of a coincidence to explain.

Theory: In US presidential elections, the more charismatic candidate wins.

People who write about politics, whether on the left or the right, have a consistent bias: they take politics seriously. When one candidate beats another they look for political explanations. The country is shifting to the left, or the right. And that sort of shift can certainly be the result of a presidential election, which makes it easy to believe it was the cause.

But when I think about why I voted for Clinton over the first George Bush, it wasn't because I was shifting to the left. Clinton just seemed more dynamic. He seemed to want the job more. Bush seemed old and tired. I suspect it was the same for a lot of voters.

Clinton didn't represent any national shift leftward. [1] He was just more charismatic than George Bush or (God help us) Bob Dole. In 2000 we practically got a controlled experiment to prove it: Gore had Clinton's policies, but not his charisma, and he suffered proportionally. [2] Same story in 2004. Kerry was smarter and more articulate than Bush, but rather a stiff. And Kerry lost.

As I looked further back, I kept finding the same pattern. Pundits said Carter beat Ford because the country distrusted the Republicans after Watergate. And yet it also happened that Carter was famous for his big grin and folksy ways, and Ford for being a boring klutz. Four years later, pundits said the country had lurched to

the right. But Reagan, a former actor, also happened to be even more charismatic than Carter (whose grin was somewhat less cheery after four stressful years in office). In 1984 the charisma gap between Reagan and Mondale was like that between Clinton and Dole, with similar results. The first George Bush managed to win in 1988, though he would later be vanquished by one of the most charismatic presidents ever, because in 1988 he was up against the notoriously uncharismatic Michael Dukakis.

These are the elections I remember personally, but apparently the same pattern played out in 1964 and 1972. The most recent counterexample appears to be 1968, when Nixon beat the more charismatic Hubert Humphrey. But when you examine that election, it tends to support the charisma theory more than contradict it. As Joe McGinnis recounts in his famous book *The Selling of the President 1968*, Nixon knew he had less charisma than Humphrey, and thus simply refused to debate him on TV. He knew he couldn't afford to let the two of them be seen side by side.

Now a candidate probably couldn't get away with refusing to debate. But in 1968 the custom of televised debates was still evolving. In effect, Nixon won in 1968 because voters were never allowed to see the real Nixon. All they saw were carefully scripted campaign spots.

Oddly enough, the most recent true counterexample is probably 1960. Though this election is usually given as an example of the power of TV, Kennedy apparently would not have won without fraud by party machines in Illinois and Texas. But TV was still young in 1960; only 87% of households had it. [3] Undoubtedly TV helped Kennedy, so historians are correct in regarding this election as a watershed. TV required a new kind of candidate. There would be no more Calvin Coolidges.

The charisma theory may also explain why Democrats tend to lose presidential elections. The core of the Democrats' ideology seems to be a belief in government. Perhaps this tends to attract people who are earnest, but dull. Dukakis, Gore, and Kerry were so similar in that respect that they might have been brothers. Good thing for the Democrats that their screen lets through an occasional Clinton, even if some scandal results. [4]

One would like to believe elections are won and lost on issues, if only fake ones like Willie Horton. And yet, if they are, we have a remarkable coincidence to explain. In every presidential election since TV became widespread, the apparently more charismatic candidate has won. Surprising, isn't it, that voters' opinions on the issues have lined up with charisma for 11 elections in a row?

The political commentators who come up with shifts to the left or right in their morning-after analyses are like the financial reporters stuck writing stories day after day about the random fluctuations of the stock market. Day ends, market closes up or down, reporter looks for good or bad news respectively, and writes that the market was up on news of Intel's earnings, or down on fears of instability in the Middle East. Suppose we could somehow feed these reporters false

information about market closes, but give them all the other news intact. Does anyone believe they would notice the anomaly, and not simply write that stocks were up (or down) on whatever good (or bad) news there was that day? That they would say, hey, wait a minute, how can stocks be up with all this unrest in the Middle East?

I'm not saying that issues don't matter to voters. Of course they do. But the major parties know so well which issues matter how much to how many voters, and adjust their message so precisely in response, that they tend to split the difference on the issues, leaving the election to be decided by the one factor they can't control: charisma.

If the Democrats had been running a candidate as charismatic as Clinton in the 2004 election, he'd have won. And we'd be reading that the election was a referendum on the war in Iraq, instead of that the Democrats are out of touch with evangelical Christians in middle America.

During the 1992 election, the Clinton campaign staff had a big sign in their office saying "It's the economy, stupid." Perhaps it was even simpler than they thought.

Postscript

Opinions seem to be divided about the charisma theory. Some say it's impossible, others say it's obvious. This seems a good sign. Perhaps it's in the sweet spot midway between.

As for it being impossible, I reply: here's the data; here's the theory; theory explains data 100%. To a scientist, at least, that means it deserves attention, however implausible it seems.

You can't believe voters are so superficial that they just choose the most charismatic guy? My theory doesn't require that. I'm not proposing that charisma is the only factor, just that it's the only one *left* after the efforts of the two parties cancel one another out.

As for the theory being obvious, as far as I know, no one has proposed it before. Election forecasters are proud when they can achieve the same results with much more complicated models.

Finally, to the people who say that the theory is probably true, but rather depressing: it's not so bad as it seems. The phenomenon is like a pricing anomaly; once people realize it's there, it will disappear. Once both parties realize it's a waste of time to nominate uncharismatic candidates, they'll tend to nominate only the most charismatic ones. And if the candidates are equally charismatic, charisma

will cancel out, and elections will be decided on issues, as political commentators like to think they are now.

Notes

[1] As Clinton himself discovered to his surprise when, in one of his first acts as president, he tried to shift the military leftward. After a bruising fight he escaped with a face-saving compromise.

[2] True, Gore won the popular vote. But politicians know the electoral vote decides the election, so that's what they campaign for. If Bush had been campaigning for the popular vote he would presumably have got more of it. (Thanks to judgmentalist for this point.)

[3] Source: Nielsen Media Research. Of the remaining 13%, 11 didn't have TV because they couldn't afford it. I'd argue that the missing 11% were probably also the 11% most susceptible to charisma.

[4] One implication of this theory is that parties shouldn't be too quick to reject candidates with skeletons in their closets. Charismatic candidates will tend to have more skeletons than squeaky clean dullards, but in practice that doesn't seem to lose elections. The current Bush, for example, probably did more drugs in his twenties than any preceding president, and yet managed to get elected with a base of evangelical Christians. All you have to do is say you've reformed, and stonewall about the details.

Thanks to Trevor Blackwell, Maria Daniels, Jessica Livingston, Jackie McDonough, and Robert Morris for reading drafts of this, and to Eric Raymond for pointing out that I was wrong about 1968.

[Comment](#) on this essay.

[What Charisma Is](#)

[Politics and the Art of Acting](#)

[Japanese Translation](#)

Made in USA

November 2004

(This is a new essay for the Japanese edition of [Hackers & Painters](). It tries to explain why Americans make some things well and others badly.)

A few years ago an Italian friend of mine travelled by train from Boston to Providence. She had only been in America for a couple weeks and hadn't seen much of the country yet. She arrived looking astonished. "It's so *ugly!"*

People from other rich countries can scarcely imagine the squalor of the man-made bits of America. In travel books they show you mostly natural environments: the Grand Canyon, whitewater rafting, horses in a field. If you see pictures with man-made things in them, it will be either a view of the New York skyline shot from a discreet distance, or a carefully cropped image of a seacoast town in Maine.

How can it be, visitors must wonder. How can the richest country in the world look like this?

Oddly enough, it may not be a coincidence. Americans are good at some things and bad at others. We're good at making movies and software, and bad at making cars and cities. And I think we may be good at what we're good at for the same reason we're bad at what we're bad at. We're impatient. In America, if you want to do something, you don't worry that it might come out badly, or upset delicate social balances, or that people might think you're getting above yourself. If you want to do something, as Nike says, *just do it.*

This works well in some fields and badly in others. I suspect it works in movies and

software because they're both messy processes. "Systematic" is the last word I'd use to describe the way good programmers write software. Code is not something they assemble painstakingly after careful planning, like the pyramids. It's something they plunge into, working fast and constantly changing their minds, like a charcoal sketch.

In software, paradoxical as it sounds, good craftsmanship means working fast. If you work slowly and meticulously, you merely end up with a very fine implementation of your initial, mistaken idea. Working slowly and meticulously is premature optimization. Better to get a prototype done fast, and see what new ideas it gives you.

It sounds like making movies works a lot like making software. Every movie is a Frankenstein, full of imperfections and usually quite different from what was originally envisioned. But interesting, and finished fairly quickly.

I think we get away with this in movies and software because they're both malleable mediums. Boldness pays. And if at the last minute two parts don't quite fit, you can figure out some hack that will at least conceal the problem.

Not so with cars, or cities. They are all too physical. If the car business worked like software or movies, you'd surpass your competitors by making a car that weighed only fifty pounds, or folded up to the size of a motorcycle when you wanted to park it. But with physical products there are more constraints. You don't win by dramatic innovations so much as by good taste and attention to detail.

The trouble is, the very word "taste" sounds slightly ridiculous to American ears. It seems pretentious, or frivolous, or even effeminate. Blue staters think it's "subjective," and red staters think it's for sissies. So anyone in America who really cares about design will be sailing upwind.

Twenty years ago we used to hear that the problem with the US car industry was the workers. We don't hear that any more now that Japanese companies are building cars in the US. The problem with American cars is bad design. You can see that just by looking at them.

All that extra sheet metal on the AMC Matador wasn't added by the workers. The problem with this car, as with American cars today, is that it was designed by marketing people instead of designers.

Why do the Japanese make better cars than us? Some say it's because their culture encourages cooperation. That may come into it. But in this case it seems more to the point that their culture prizes design and craftsmanship.

For centuries the Japanese have made finer things than we have in the West. When you look at swords they made in 1200, you just can't believe the date on the

label is right. Presumably their cars fit together more precisely than ours for the same reason their joinery always has. They're obsessed with making things well.

Not us. When we make something in America, our aim is just to get the job done. Once we reach that point, we take one of two routes. We can stop there, and have something crude but serviceable, like a Vise-grip. Or we can improve it, which usually means encrusting it with gratuitous ornament. When we want to make a car "better," we stick [tail fins](#) on it, or make it [longer,](#) or make the [windows smaller,](#) depending on the current fashion.

Ditto for houses. In America you can have either a flimsy box banged together out of two by fours and drywall, or a McMansion-- a flimsy box banged together out of two by fours and drywall, but larger, more dramatic-looking, and full of expensive fittings. Rich people don't get better design or craftsmanship; they just get a larger, more conspicuous version of the standard house.

We don't especially prize design or craftsmanship here. What we like is speed, and we're willing to do something in an ugly way to get it done fast. In some fields, like software or movies, this is a net win.

But it's not just that software and movies are malleable mediums. In those businesses, the designers (though they're not generally called that) have more power. Software companies, at least successful ones, tend to be run by programmers. And in the film industry, though producers may second-guess directors, the director controls most of what appears on the screen. And so American software and movies, and Japanese cars, all have this in common: the people in charge care about design-- the former because the designers are in charge, and the latter because the whole culture cares about design.

I think most Japanese executives would be horrified at the idea of making a bad car. Whereas American executives, in their hearts, still believe the most important thing about a car is the image it projects. Make a good car? What's "good?" It's so *subjective.* If you want to know how to design a car, ask a focus group.

Instead of relying on their own internal design compass (like Henry Ford did), American car companies try to make what marketing people think consumers want. But it isn't working. American cars continue to lose market share. And the reason is that the customer doesn't want what he thinks he wants.

Letting focus groups design your cars for you only wins in the short term. In the long term, it pays to bet on good design. The focus group may say they want the meretricious feature du jour, but what they want even more is to imitate sophisticated buyers, and they, though a small minority, really do care about good design. Eventually the pimps and drug dealers notice that the doctors and lawyers have switched from Cadillac to Lexus, and do the same.

Apple is an interesting counterexample to the general American trend. If you want to buy a nice CD player, you'll probably buy a Japanese one. But if you want to buy

an MP3 player, you'll probably buy an iPod. What happened? Why doesn't Sony dominate MP3 players? Because Apple is in the consumer electronics business now, and unlike other American companies, they're obsessed with good design. Or more precisely, their CEO is.

I just got an iPod, and it's not just nice. It's *surprisingly* nice. For it to surprise me, it must be satisfying expectations I didn't know I had. No focus group is going to discover those. Only a great designer can.

Cars aren't the worst thing we make in America. Where the just-do-it model fails most dramatically is in our cities-- or rather, exurbs. If real estate developers operated on a large enough scale, if they built whole towns, market forces would compel them to build towns that didn't suck. But they only build a couple office buildings or suburban streets at a time, and the result is so depressing that the inhabitants consider it a great treat to fly to Europe and spend a couple weeks living what is, for people there, just everyday life. [1]

But the just-do-it model does have advantages. It seems the clear winner for generating wealth and technical innovations (which are practically the same thing). I think speed is the reason. It's hard to create wealth by making a commodity. The real value is in things that are new, and if you want to be the first to make something, it helps to work fast. For better or worse, the just-do-it model is fast, whether you're Dan Bricklin writing the prototype of VisiCalc in a weekend, or a real estate developer building a block of shoddy condos in a month.

If I had to choose between the just-do-it model and the careful model, I'd probably choose just-do-it. But do we have to choose? Could we have it both ways? Could Americans have nice places to live without undermining the impatient, individualistic spirit that makes us good at software? Could other countries introduce more individualism into their technology companies and research labs without having it metastasize as strip malls? I'm optimistic. It's harder to say about other countries, but in the US, at least, I think we can have both.

Apple is an encouraging example. They've managed to preserve enough of the impatient, hackerly spirit you need to write software. And yet when you pick up a new Apple laptop, well, it doesn't seem American. It's too perfect. It seems as if it must have been made by a Swedish or a Japanese company.

In many technologies, version 2 has higher resolution. Why not in design generally? I think we'll gradually see national characters superseded by occupational characters: hackers in Japan will be allowed to behave with a willfulness that would now seem unJapanese, and products in America will be designed with an insistence on taste that would now seem unAmerican. Perhaps the most successful countries, in the future, will be those most willing to ignore what are now considered national characters, and do each kind of work in the way that works best. Race you.

Notes

[1] Japanese cities are ugly too, but for different reasons. Japan is prone to earthquakes, so buildings are traditionally seen as temporary; there is no grand tradition of city planning like the one Europeans inherited from Rome. The other cause is the notoriously corrupt relationship between the government and construction companies.

·

American Gothic

·

The John Rain Books

What You'll Wish You'd Known

January 2005

(I wrote this talk for a high school. I never actually gave it, because the school authorities vetoed the plan to invite me.)

When I said I was speaking at a high school, my friends were curious. What will you say to high school students? So I asked them, what do you wish someone had told you in high school? Their answers were remarkably similar. So I'm going to tell you what we all wish someone had told us.

I'll start by telling you something you don't have to know in high school: what you want to do with your life. People are always asking you this, so you think you're supposed to have an answer. But adults ask this mainly as a conversation starter. They want to know what sort of person you are, and this question is just to get you talking. They ask it the way you might poke a hermit crab in a tide pool, to see what it does.

If I were back in high school and someone asked about my plans, I'd say that my first priority was to learn what the options were. You don't need to be in a rush to choose your life's work. What you need to do is discover what you like. You have to work on stuff you like if you want to be good at what you do.

It might seem that nothing would be easier than deciding what you like, but it turns out to be hard, partly because it's hard to get an accurate picture of most jobs. Being a doctor is not the way it's portrayed on TV. Fortunately you can also watch real doctors, by volunteering in hospitals. [1]

But there are other jobs you can't learn about, because no one is doing them yet. Most of the work I've done in the last ten years didn't exist when I was in high school. The world changes fast, and the rate at which it changes is itself speeding up. In such a world it's not a good idea to have fixed plans.

And yet every May, speakers all over the country fire up the Standard Graduation Speech, the theme of which is: don't give up on your dreams. I know what they mean, but this is a bad way to put it, because it implies you're supposed to be bound by some plan you made early on. The computer world has a name for this: premature optimization. And it is synonymous with disaster. These speakers would do better to say simply, don't give up.

What they really mean is, don't get demoralized. Don't think that you can't do what other people can. And I agree you shouldn't underestimate your potential. People who've done great things tend to seem as if they were a race apart. And most biographies only exaggerate this illusion, partly due to the worshipful attitude biographers inevitably sink into, and partly because, knowing how the story ends, they can't help streamlining the plot till it seems like the subject's life was a matter of destiny, the mere unfolding of some innate genius. In fact I suspect if you had the sixteen year old Shakespeare or Einstein in school with you, they'd seem impressive, but not totally unlike your other friends.

Which is an uncomfortable thought. If they were just like us, then they had to work very hard to do what they did. And that's one reason we like to believe in genius. It gives us an excuse for being lazy. If these guys were able to do what they did only because of some magic Shakespeareness or Einsteinness, then it's not our fault if we can't do something as good.

I'm not saying there's no such thing as genius. But if you're trying to choose between two theories and one gives you an excuse for being lazy, the other one is probably right.

So far we've cut the Standard Graduation Speech down from "don't give up on your dreams" to "what someone else can do, you can do." But it needs to be cut still further. There is *some* variation in natural ability. Most people overestimate its role, but it does exist. If I were talking to a guy four feet tall whose ambition was to play in the NBA, I'd feel pretty stupid saying, you can do anything if you really try. [2]

We need to cut the Standard Graduation Speech down to, "what someone else with your abilities can do, you can do; and don't underestimate your abilities." But as so often happens, the closer you get to the truth, the messier your sentence gets. We've taken a nice, neat (but wrong) slogan, and churned it up like a mud puddle. It doesn't make a very good speech anymore. But worse still, it doesn't tell you what to do anymore. Someone with your abilities? What are your abilities?

Upwind

I think the solution is to work in the other direction. Instead of working back from a goal, work forward from promising situations. This is what most successful people actually do anyway.

In the graduation-speech approach, you decide where you want to be in twenty years, and then ask: what should I do now to get there? I propose instead that you don't commit to anything in the future, but just look at the options available now, and choose those that will give you the most promising range of options afterward.

It's not so important what you work on, so long as you're not wasting your time.

Work on things that interest you and increase your options, and worry later about which you'll take.

Suppose you're a college freshman deciding whether to major in math or economics. Well, math will give you more options: you can go into almost any field from math. If you major in math it will be easy to get into grad school in economics, but if you major in economics it will be hard to get into grad school in math.

Flying a glider is a good metaphor here. Because a glider doesn't have an engine, you can't fly into the wind without losing a lot of altitude. If you let yourself get far downwind of good places to land, your options narrow uncomfortably. As a rule you want to stay upwind. So I propose that as a replacement for "don't give up on your dreams." Stay upwind.

How do you do that, though? Even if math is upwind of economics, how are you supposed to know that as a high school student?

Well, you don't, and that's what you need to find out. Look for smart people and hard problems. Smart people tend to clump together, and if you can find such a clump, it's probably worthwhile to join it. But it's not straightforward to find these, because there is a lot of faking going on.

To a newly arrived undergraduate, all university departments look much the same. The professors all seem forbiddingly intellectual and publish papers unintelligible to outsiders. But while in some fields the papers are unintelligible because they're full of hard ideas, in others they're deliberately written in an obscure way to seem as if they're saying something important. This may seem a scandalous proposition, but it has been experimentally verified, in the famous *Social Text* affair. Suspecting that the papers published by literary theorists were often just intellectual-sounding nonsense, a physicist deliberately wrote a paper full of intellectual-sounding nonsense, and submitted it to a literary theory journal, which published it.

The best protection is always to be working on hard problems. Writing novels is hard. Reading novels isn't. Hard means worry: if you're not worrying that something you're making will come out badly, or that you won't be able to understand something you're studying, then it isn't hard enough. There has to be suspense.

Well, this seems a grim view of the world, you may think. What I'm telling you is that you should worry? Yes, but it's not as bad as it sounds. It's exhilarating to overcome worries. You don't see faces much happier than people winning gold medals. And you know why they're so happy? Relief.

I'm not saying this is the only way to be happy. Just that some kinds of worry are not as bad as they sound.

Ambition

In practice, "stay upwind" reduces to "work on hard problems." And you can start today. I wish I'd grasped that in high school.

Most people like to be good at what they do. In the so-called real world this need is a powerful force. But high school students rarely benefit from it, because they're given a fake thing to do. When I was in high school, I let myself believe that my job was to be a high school student. And so I let my need to be good at what I did be satisfied by merely doing well in school.

If you'd asked me in high school what the difference was between high school kids and adults, I'd have said it was that adults had to earn a living. Wrong. It's that adults take responsibility for themselves. Making a living is only a small part of it. Far more important is to take intellectual responsibility for oneself.

If I had to go through high school again, I'd treat it like a day job. I don't mean that I'd slack in school. Working at something as a day job doesn't mean doing it badly. It means not being defined by it. I mean I wouldn't think of myself as a high school student, just as a musician with a day job as a waiter doesn't think of himself as a waiter. [3] And when I wasn't working at my day job I'd start trying to do real work.

When I ask people what they regret most about high school, they nearly all say the same thing: that they wasted so much time. If you're wondering what you're doing now that you'll regret most later, that's probably it. [4]

Some people say this is inevitable — that high school students aren't capable of getting anything done yet. But I don't think this is true. And the proof is that you're bored. You probably weren't bored when you were eight. When you're eight it's called "playing" instead of "hanging out," but it's the same thing. And when I was eight, I was rarely bored. Give me a back yard and a few other kids and I could play all day.

The reason this got stale in middle school and high school, I now realize, is that I was ready for something else. Childhood was getting old.

I'm not saying you shouldn't hang out with your friends — that you should all become humorless little robots who do nothing but work. Hanging out with friends is like chocolate cake. You enjoy it more if you eat it occasionally than if you eat nothing but chocolate cake for every meal. No matter how much you like chocolate cake, you'll be pretty queasy after the third meal of it. And that's what the malaise one feels in high school is: mental queasiness. [5]

You may be thinking, we have to do more than get good grades. We have to have *extracurricular activities.* But you know perfectly well how bogus most of these are. Collecting donations for a charity is an admirable thing to do, but it's not *hard.* It's not getting something done. What I mean by getting something done is learning how to write well, or how to program computers, or what life was really like in

preindustrial societies, or how to draw the human face from life. This sort of thing rarely translates into a line item on a college application.

Corruption

It's dangerous to design your life around getting into college, because the people you have to impress to get into college are not a very discerning audience. At most colleges, it's not the professors who decide whether you get in, but admissions officers, and they are nowhere near as smart. They're the NCOs of the intellectual world. They can't tell how smart you are. The mere existence of prep schools is proof of that.

Few parents would pay so much for their kids to go to a school that didn't improve their admissions prospects. Prep schools openly say this is one of their aims. But what that means, if you stop to think about it, is that they can hack the admissions process: that they can take the very same kid and make him seem a more appealing candidate than he would if he went to the local public school. [6]

Right now most of you feel your job in life is to be a promising college applicant. But that means you're designing your life to satisfy a process so mindless that there's a whole industry devoted to subverting it. No wonder you become cynical. The malaise you feel is the same that a producer of reality TV shows or a tobacco industry executive feels. And you don't even get paid a lot.

So what do you do? What you should not do is rebel. That's what I did, and it was a mistake. I didn't realize exactly what was happening to us, but I smelled a major rat. And so I just gave up. Obviously the world sucked, so why bother?

When I discovered that one of our teachers was herself using Cliff's Notes, it seemed par for the course. Surely it meant nothing to get a good grade in such a class.

In retrospect this was stupid. It was like someone getting fouled in a soccer game and saying, hey, you fouled me, that's against the rules, and walking off the field in indignation. Fouls happen. The thing to do when you get fouled is not to lose your cool. Just keep playing.

By putting you in this situation, society has fouled you. Yes, as you suspect, a lot of the stuff you learn in your classes is crap. And yes, as you suspect, the college admissions process is largely a charade. But like many fouls, this one was unintentional. [7] So just keep playing.

Rebellion is almost as stupid as obedience. In either case you let yourself be defined by what they tell you to do. The best plan, I think, is to step onto an orthogonal vector. Don't just do what they tell you, and don't just refuse to. Instead treat school as a day job. As day jobs go, it's pretty sweet. You're done at 3 o'clock, and you can even work on your own stuff while you're there.

Curiosity

And what's your real job supposed to be? Unless you're Mozart, your first task is to figure that out. What are the great things to work on? Where are the imaginative people? And most importantly, what are you interested in? The word "aptitude" is misleading, because it implies something innate. The most powerful sort of aptitude is a consuming interest in some question, and such interests are often acquired tastes.

A distorted version of this idea has filtered into popular culture under the name "passion." I recently saw an ad for waiters saying they wanted people with a "passion for service." The real thing is not something one could have for waiting on tables. And passion is a bad word for it. A better name would be curiosity.

Kids are curious, but the curiosity I mean has a different shape from kid curiosity. Kid curiosity is broad and shallow; they ask why at random about everything. In most adults this curiosity dries up entirely. It has to: you can't get anything done if you're always asking why about everything. But in ambitious adults, instead of drying up, curiosity becomes narrow and deep. The mud flat morphs into a well.

Curiosity turns work into play. For Einstein, relativity wasn't a book full of hard stuff he had to learn for an exam. It was a mystery he was trying to solve. So it probably felt like less work to him to invent it than it would seem to someone now to learn it in a class.

One of the most dangerous illusions you get from school is the idea that doing great things requires a lot of discipline. Most subjects are taught in such a boring way that it's only by discipline that you can flog yourself through them. So I was surprised when, early in college, I read a quote by Wittgenstein saying that he had no self-discipline and had never been able to deny himself anything, not even a cup of coffee.

Now I know a number of people who do great work, and it's the same with all of them. They have little discipline. They're all terrible procrastinators and find it almost impossible to make themselves do anything they're not interested in. One still hasn't sent out his half of the thank-you notes from his wedding, four years ago. Another has 26,000 emails in her inbox.

I'm not saying you can get away with zero self-discipline. You probably need about the amount you need to go running. I'm often reluctant to go running, but once I do, I enjoy it. And if I don't run for several days, I feel ill. It's the same with people who do great things. They know they'll feel bad if they don't work, and they have enough discipline to get themselves to their desks to start working. But once they get started, interest takes over, and discipline is no longer necessary.

Do you think Shakespeare was gritting his teeth and diligently trying to write Great Literature? Of course not. He was having fun. That's why he's so good.

If you want to do good work, what you need is a great curiosity about a promising question. The critical moment for Einstein was when he looked at Maxwell's equations and said, what the hell is going on here?

It can take years to zero in on a productive question, because it can take years to figure out what a subject is really about. To take an extreme example, consider math. Most people think they hate math, but the boring stuff you do in school under the name "mathematics" is not at all like what mathematicians do.

The great mathematician G. H. Hardy said he didn't like math in high school either. He only took it up because he was better at it than the other students. Only later did he realize math was interesting — only later did he start to ask questions instead of merely answering them correctly.

When a friend of mine used to grumble because he had to write a paper for school, his mother would tell him: find a way to make it interesting. That's what you need to do: find a question that makes the world interesting. People who do great things look at the same world everyone else does, but notice some odd detail that's compellingly mysterious.

And not only in intellectual matters. Henry Ford's great question was, why do cars have to be a luxury item? What would happen if you treated them as a commodity? Franz Beckenbauer's was, in effect, why does everyone have to stay in his position? Why can't defenders score goals too?

Now

If it takes years to articulate great questions, what do you do now, at sixteen? Work toward finding one. Great questions don't appear suddenly. They gradually congeal in your head. And what makes them congeal is experience. So the way to find great questions is not to search for them — not to wander about thinking, what great discovery shall I make? You can't answer that; if you could, you'd have made it.

The way to get a big idea to appear in your head is not to hunt for big ideas, but to put in a lot of time on work that interests you, and in the process keep your mind open enough that a big idea can take roost. Einstein, Ford, and Beckenbauer all used this recipe. They all knew their work like a piano player knows the keys. So when something seemed amiss to them, they had the confidence to notice it.

Put in time how and on what? Just pick a project that seems interesting: to master some chunk of material, or to make something, or to answer some question. Choose a project that will take less than a month, and make it something you have the means to finish. Do something hard enough to stretch you, but only just, especially at first. If you're deciding between two projects, choose whichever seems most fun. If one blows up in your face, start another. Repeat till, like an internal combustion engine, the process becomes self-sustaining, and each project generates the next one. (This could take years.)

It may be just as well not to do a project "for school," if that will restrict you or make it seem like work. Involve your friends if you want, but not too many, and only if they're not flakes. Friends offer moral support (few startups are started by one person), but secrecy also has its advantages. There's something pleasing about a secret project. And you can take more risks, because no one will know if you fail.

Don't worry if a project doesn't seem to be on the path to some goal you're supposed to have. Paths can bend a lot more than you think. So let the path grow out the project. The most important thing is to be excited about it, because it's by doing that you learn.

Don't disregard unseemly motivations. One of the most powerful is the desire to be better than other people at something. Hardy said that's what got him started, and I think the only unusual thing about him is that he admitted it. Another powerful motivator is the desire to do, or know, things you're not supposed to. Closely related is the desire to do something audacious. Sixteen year olds aren't supposed to write novels. So if you try, anything you achieve is on the plus side of the ledger; if you fail utterly, you're doing no worse than expectations. [8]

Beware of bad models. Especially when they excuse laziness. When I was in high school I used to write "existentialist" short stories like ones I'd seen by famous writers. My stories didn't have a lot of plot, but they were very deep. And they were less work to write than entertaining ones would have been. I should have known that was a danger sign. And in fact I found my stories pretty boring; what excited me was the idea of writing serious, intellectual stuff like the famous writers.

Now I have enough experience to realize that those famous writers actually sucked. Plenty of famous people do; in the short term, the quality of one's work is only a small component of fame. I should have been less worried about doing something that seemed cool, and just done something I liked. That's the actual road to coolness anyway.

A key ingredient in many projects, almost a project on its own, is to find good books. Most books are bad. Nearly all textbooks are bad. [9] So don't assume a subject is to be learned from whatever book on it happens to be closest. You have to search actively for the tiny number of good books.

The important thing is to get out there and do stuff. Instead of waiting to be taught, go out and learn.

Your life doesn't have to be shaped by admissions officers. It could be shaped by your own curiosity. It is for all ambitious adults. And you don't have to wait to start. In fact, you don't have to wait to be an adult. There's no switch inside you that magically flips when you turn a certain age or graduate from some institution. You start being an adult when you decide to take responsibility for your life. You

can do that at any age. [10]

This may sound like bullshit. I'm just a minor, you may think, I have no money, I have to live at home, I have to do what adults tell me all day long. Well, most adults labor under restrictions just as cumbersome, and they manage to get things done. If you think it's restrictive being a kid, imagine having kids.

The only real difference between adults and high school kids is that adults realize they need to get things done, and high school kids don't. That realization hits most people around 23. But I'm letting you in on the secret early. So get to work. Maybe you can be the first generation whose greatest regret from high school isn't how much time you wasted.

Notes

[1] A doctor friend warns that even this can give an inaccurate picture. "Who knew how much time it would take up, how little autonomy one would have for endless years of training, and how unbelievably annoying it is to carry a beeper?"

[2] His best bet would probably be to become dictator and intimidate the NBA into letting him play. So far the closest anyone has come is Secretary of Labor.

[3] A day job is one you take to pay the bills so you can do what you really want, like play in a band, or invent relativity.

Treating high school as a day job might actually make it easier for some students to get good grades. If you treat your classes as a game, you won't be demoralized if they seem pointless.

However bad your classes, you need to get good grades in them to get into a decent college. And that *is* worth doing, because universities are where a lot of the clumps of smart people are these days.

[4] The second biggest regret was caring so much about unimportant things. And especially about what other people thought of them.

I think what they really mean, in the latter case, is caring what random people thought of them. Adults care just as much what other people think, but they get to be more selective about the other people.

I have about thirty friends whose opinions I care about, and the opinion of the rest of the world barely affects me. The problem in high school is that your peers are chosen for you by accidents of age and geography, rather than by you based on respect for their judgement.

[5] The key to wasting time is distraction. Without distractions it's too obvious to your brain that you're not doing anything with it, and you start to feel uncomfortable. If you want to measure how dependent you've become on distractions, try this experiment: set aside a chunk of time on a weekend and sit alone and think. You can have a notebook to write your thoughts down in, but nothing else: no friends, TV, music, phone, IM, email, Web, games, books, newspapers, or magazines. Within an hour most people will feel a strong craving for distraction.

[6] I don't mean to imply that the only function of prep schools is to trick admissions officers. They also generally provide a better education. But try this thought experiment: suppose prep schools supplied the same superior education but had a tiny (.001) negative effect on college admissions. How many parents would still send their kids to them?

It might also be argued that kids who went to prep schools, because they've learned more, *are* better college candidates. But this seems empirically false. What you learn in even the best high school is rounding error compared to what you learn in college. Public school kids arrive at college with a slight disadvantage, but they start to pull ahead in the sophomore year.

(I'm not saying public school kids are smarter than preppies, just that they are within any given college. That follows necessarily if you agree prep schools improve kids' admissions prospects.)

[7] Why does society foul you? Indifference, mainly. There are simply no outside forces pushing high school to be good. The air traffic control system works because planes would crash otherwise. Businesses have to deliver because otherwise competitors would take their customers. But no planes crash if your school sucks, and it has no competitors. High school isn't evil; it's random; but random is pretty bad.

[8] And then of course there is money. It's not a big factor in high school, because you can't do much that anyone wants. But a lot of great things were created mainly to make money. Samuel Johnson said "no man but a blockhead ever wrote except for money." (Many hope he was exaggerating.)

[9] Even college textbooks are bad. When you get to college, you'll find that (with a few stellar exceptions) the textbooks are not written by the leading scholars in the field they describe. Writing college textbooks is unpleasant work, done mostly by people who need the money. It's unpleasant because the publishers exert so much control, and there are few things worse than close supervision by someone who doesn't understand what you're doing. This phenomenon is apparently even worse in the production of high school textbooks.

[10] Your teachers are always telling you to behave like adults. I wonder if they'd like it if you did. You may be loud and disorganized, but you're very docile

compared to adults. If you actually started acting like adults, it would be just as if a bunch of adults had been transposed into your bodies. Imagine the reaction of an FBI agent or taxi driver or reporter to being told they had to ask permission to go the bathroom, and only one person could go at a time. To say nothing of the things you're taught. If a bunch of actual adults suddenly found themselves trapped in high school, the first thing they'd do is form a union and renegotiate all the rules with the administration.

.

[Why Nerds are Unpopular](#)

.

[Japanese Translation](#)

.

[Russian Translation](#)

.

[Georgian Translation](#)

How to Start a Startup

March 2005

(This essay is derived from a talk at the Harvard Computer Society.)

You need three things to create a successful startup: to start with good people, to make something customers actually want, and to spend as little money as possible. Most startups that fail do it because they fail at one of these. A startup that does all three will probably succeed.

And that's kind of exciting, when you think about it, because all three are doable. Hard, but doable. And since a startup that succeeds ordinarily makes its founders rich, that implies getting rich is doable too. Hard, but doable.

If there is one message I'd like to get across about startups, that's it. There is no magically difficult step that requires brilliance to solve.

The Idea

In particular, you don't need a brilliant idea to start a startup around. The way a startup makes money is to offer people better technology than they have now. But what people have now is often so bad that it doesn't take brilliance to do better.

Google's plan, for example, was simply to create a search site that didn't suck. They had three new ideas: index more of the Web, use links to rank search results, and have clean, simple web pages with unintrusive keyword-based ads. Above all, they were determined to make a site that was good to use. No doubt there are great technical tricks within Google, but the overall plan was straightforward. And while they probably have bigger ambitions now, this alone brings them a billion dollars a year. [1]

There are plenty of other areas that are just as backward as search was before Google. I can think of several heuristics for generating ideas for startups, but most reduce to this: look at something people are trying to do, and figure out how to do it in a way that doesn't suck.

For example, dating sites currently suck far worse than search did before Google. They all use the same simple-minded model. They seem to have approached the

problem by thinking about how to do database matches instead of how dating works in the real world. An undergrad could build something better as a class project. And yet there's a lot of money at stake. Online dating is a valuable business now, and it might be worth a hundred times as much if it worked.

An idea for a startup, however, is only a beginning. A lot of would-be startup founders think the key to the whole process is the initial idea, and from that point all you have to do is execute. Venture capitalists know better. If you go to VC firms with a brilliant idea that you'll tell them about if they sign a nondisclosure agreement, most will tell you to get lost. That shows how much a mere idea is worth. The market price is less than the inconvenience of signing an NDA.

Another sign of how little the initial idea is worth is the number of startups that change their plan en route. Microsoft's original plan was to make money selling programming languages, of all things. Their current business model didn't occur to them until IBM dropped it in their lap five years later.

Ideas for startups are worth something, certainly, but the trouble is, they're not transferrable. They're not something you could hand to someone else to execute. Their value is mainly as starting points: as questions for the people who had them to continue thinking about.

What matters is not ideas, but the people who have them. Good people can fix bad ideas, but good ideas can't save bad people.

People

What do I mean by good people? One of the best tricks I learned during our startup was a rule for deciding who to hire. Could you describe the person as an animal? It might be hard to translate that into another language, but I think everyone in the US knows what it means. It means someone who takes their work a little too seriously; someone who does what they do so well that they pass right through professional and cross over into obsessive.

What it means specifically depends on the job: a salesperson who just won't take no for an answer; a hacker who will stay up till 4:00 AM rather than go to bed leaving code with a bug in it; a PR person who will cold-call *New York Times* reporters on their cell phones; a graphic designer who feels physical pain when something is two millimeters out of place.

Almost everyone who worked for us was an animal at what they did. The woman in charge of sales was so tenacious that I used to feel sorry for potential customers on the phone with her. You could sense them squirming on the hook, but you knew there would be no rest for them till they'd signed up.

If you think about people you know, you'll find the animal test is easy to apply. Call the person's image to mind and imagine the sentence "so-and-so is an animal." If you laugh, they're not. You don't need or perhaps even want this quality in big

companies, but you need it in a startup.

For programmers we had three additional tests. Was the person genuinely smart? If so, could they actually get things done? And finally, since a few good hackers have unbearable personalities, could we stand to have them around?

That last test filters out surprisingly few people. We could bear any amount of nerdiness if someone was truly smart. What we couldn't stand were people with a lot of attitude. But most of those weren't truly smart, so our third test was largely a restatement of the first.

When nerds are unbearable it's usually because they're trying too hard to seem smart. But the smarter they are, the less pressure they feel to act smart. So as a rule you can recognize genuinely smart people by their ability to say things like "I don't know," "Maybe you're right," and "I don't understand x well enough."

This technique doesn't always work, because people can be influenced by their environment. In the MIT CS department, there seems to be a tradition of acting like a brusque know-it-all. I'm told it derives ultimately from Marvin Minsky, in the same way the classic airline pilot manner is said to derive from Chuck Yeager. Even genuinely smart people start to act this way there, so you have to make allowances.

It helped us to have Robert Morris, who is one of the readiest to say "I don't know" of anyone I've met. (At least, he was before he became a professor at MIT.) No one dared put on attitude around Robert, because he was obviously smarter than they were and yet had zero attitude himself.

Like most startups, ours began with a group of friends, and it was through personal contacts that we got most of the people we hired. This is a crucial difference between startups and big companies. Being friends with someone for even a couple days will tell you more than companies could ever learn in interviews. [2]

It's no coincidence that startups start around universities, because that's where smart people meet. It's not what people learn in classes at MIT and Stanford that has made technology companies spring up around them. They could sing campfire songs in the classes so long as admissions worked the same.

If you start a startup, there's a good chance it will be with people you know from college or grad school. So in theory you ought to try to make friends with as many smart people as you can in school, right? Well, no. Don't make a conscious effort to schmooze; that doesn't work well with hackers.

What you should do in college is work on your own projects. Hackers should do this even if they don't plan to start startups, because it's the only real way to learn how to program. In some cases you may collaborate with other students, and this is the best way to get to know good hackers. The project may even grow into a

startup. But once again, I wouldn't aim too directly at either target. Don't force things; just work on stuff you like with people you like.

Ideally you want between two and four founders. It would be hard to start with just one. One person would find the moral weight of starting a company hard to bear. Even Bill Gates, who seems to be able to bear a good deal of moral weight, had to have a co-founder. But you don't want so many founders that the company starts to look like a group photo. Partly because you don't need a lot of people at first, but mainly because the more founders you have, the worse disagreements you'll have. When there are just two or three founders, you know you have to resolve disputes immediately or perish. If there are seven or eight, disagreements can linger and harden into factions. You don't want mere voting; you need unanimity.

In a technology startup, which most startups are, the founders should include technical people. During the Internet Bubble there were a number of startups founded by business people who then went looking for hackers to create their product for them. This doesn't work well. Business people are bad at deciding what to do with technology, because they don't know what the options are, or which kinds of problems are hard and which are easy. And when business people try to hire hackers, they can't tell which ones are good. Even other hackers have a hard time doing that. For business people it's roulette.

Do the founders of a startup have to include business people? That depends. We thought so when we started ours, and we asked several people who were said to know about this mysterious thing called "business" if they would be the president. But they all said no, so I had to do it myself. And what I discovered was that business was no great mystery. It's not something like physics or medicine that requires extensive study. You just try to get people to pay you for stuff.

I think the reason I made such a mystery of business was that I was disgusted by the idea of doing it. I wanted to work in the pure, intellectual world of software, not deal with customers' mundane problems. People who don't want to get dragged into some kind of work often develop a protective incompetence at it. Paul Erdos was particularly good at this. By seeming unable even to cut a grapefruit in half (let alone go to the store and buy one), he forced other people to do such things for him, leaving all his time free for math. Erdos was an extreme case, but most husbands use the same trick to some degree.

Once I was forced to discard my protective incompetence, I found that business was neither so hard nor so boring as I feared. There are esoteric areas of business that are quite hard, like tax law or the pricing of derivatives, but you don't need to know about those in a startup. All you need to know about business to run a startup are commonsense things people knew before there were business schools, or even universities.

If you work your way down the Forbes 400 making an x next to the name of each person with an MBA, you'll learn something important about business school. After

Warren Buffett, you don't hit another MBA till number 22, Phil Knight, the CEO of Nike. There are only 5 MBAs in the top 50. What you notice in the Forbes 400 are a lot of people with technical backgrounds. Bill Gates, Steve Jobs, Larry Ellison, Michael Dell, Jeff Bezos, Gordon Moore. The rulers of the technology business tend to come from technology, not business. So if you want to invest two years in something that will help you succeed in business, the evidence suggests you'd do better to learn how to hack than get an MBA. [3]

There is one reason you might want to include business people in a startup, though: because you have to have at least one person willing and able to focus on what customers want. Some believe only business people can do this-- that hackers can implement software, but not design it. That's nonsense. There's nothing about knowing how to program that prevents hackers from understanding users, or about not knowing how to program that magically enables business people to understand them.

If you can't understand users, however, you should either learn how or find a co-founder who can. That is the single most important issue for technology startups, and the rock that sinks more of them than anything else.

What Customers Want

It's not just startups that have to worry about this. I think most businesses that fail do it because they don't give customers what they want. Look at restaurants. A large percentage fail, about a quarter in the first year. But can you think of one restaurant that had really good food and went out of business?

Restaurants with great food seem to prosper no matter what. A restaurant with great food can be expensive, crowded, noisy, dingy, out of the way, and even have bad service, and people will keep coming. It's true that a restaurant with mediocre food can sometimes attract customers through gimmicks. But that approach is very risky. It's more straightforward just to make the food good.

It's the same with technology. You hear all kinds of reasons why startups fail. But can you think of one that had a massively popular product and still failed?

In nearly every failed startup, the real problem was that customers didn't want the product. For most, the cause of death is listed as "ran out of funding," but that's only the immediate cause. Why couldn't they get more funding? Probably because the product was a dog, or never seemed likely to be done, or both.

When I was trying to think of the things every startup needed to do, I almost included a fourth: get a version 1 out as soon as you can. But I decided not to, because that's implicit in making something customers want. The only way to make something customers want is to get a prototype in front of them and refine it based on their reactions.

The other approach is what I call the "Hail Mary" strategy. You make elaborate

plans for a product, hire a team of engineers to develop it (people who do this tend to use the term "engineer" for hackers), and then find after a year that you've spent two million dollars to develop something no one wants. This was not uncommon during the Bubble, especially in companies run by business types, who thought of software development as something terrifying that therefore had to be carefully planned.

We never even considered that approach. As a Lisp hacker, I come from the tradition of rapid prototyping. I would not claim (at least, not here) that this is the right way to write every program, but it's certainly the right way to write software for a startup. In a startup, your initial plans are almost certain to be wrong in some way, and your first priority should be to figure out where. The only way to do that is to try implementing them.

Like most startups, we changed our plan on the fly. At first we expected our customers to be Web consultants. But it turned out they didn't like us, because our software was easy to use and we hosted the site. It would be too easy for clients to fire them. We also thought we'd be able to sign up a lot of catalog companies, because selling online was a natural extension of their existing business. But in 1996 that was a hard sell. The middle managers we talked to at catalog companies saw the Web not as an opportunity, but as something that meant more work for them.

We did get a few of the more adventurous catalog companies. Among them was Frederick's of Hollywood, which gave us valuable experience dealing with heavy loads on our servers. But most of our users were small, individual merchants who saw the Web as an opportunity to build a business. Some had retail stores, but many only existed online. And so we changed direction to focus on these users. Instead of concentrating on the features Web consultants and catalog companies would want, we worked to make the software easy to use.

I learned something valuable from that. It's worth trying very, very hard to make technology easy to use. Hackers are so used to computers that they have no idea how horrifying software seems to normal people. Stephen Hawking's editor told him that every equation he included in his book would cut sales in half. When you work on making technology easier to use, you're riding that curve up instead of down. A 10% improvement in ease of use doesn't just increase your sales 10%. It's more likely to double your sales.

How do you figure out what customers want? Watch them. One of the best places to do this was at trade shows. Trade shows didn't pay as a way of getting new customers, but they were worth it as market research. We didn't just give canned presentations at trade shows. We used to show people how to build real, working stores. Which meant we got to watch as they used our software, and talk to them about what they needed.

No matter what kind of startup you start, it will probably be a stretch for you, the founders, to understand what users want. The only kind of software you can build

without studying users is the sort for which you are the typical user. But this is just the kind that tends to be open source: operating systems, programming languages, editors, and so on. So if you're developing technology for money, you're probably not going to be developing it for people like you. Indeed, you can use this as a way to generate ideas for startups: what do people who are not like you want from technology?

When most people think of startups, they think of companies like Apple or Google. Everyone knows these, because they're big consumer brands. But for every startup like that, there are twenty more that operate in niche markets or live quietly down in the infrastructure. So if you start a successful startup, odds are you'll start one of those.

Another way to say that is, if you try to start the kind of startup that has to be a big consumer brand, the odds against succeeding are steeper. The best odds are in niche markets. Since startups make money by offering people something better than they had before, the best opportunities are where things suck most. And it would be hard to find a place where things suck more than in corporate IT departments. You would not believe the amount of money companies spend on software, and the crap they get in return. This imbalance equals opportunity.

If you want ideas for startups, one of the most valuable things you could do is find a middle-sized non-technology company and spend a couple weeks just watching what they do with computers. Most good hackers have no more idea of the horrors perpetrated in these places than rich Americans do of what goes on in Brazilian slums.

Start by writing software for smaller companies, because it's easier to sell to them. It's worth so much to sell stuff to big companies that the people selling them the crap they currently use spend a lot of time and money to do it. And while you can outhack Oracle with one frontal lobe tied behind your back, you can't outsell an Oracle salesman. So if you want to win through better technology, aim at smaller customers. [4]

They're the more strategically valuable part of the market anyway. In technology, the low end always eats the high end. It's easier to make an inexpensive product more powerful than to make a powerful product cheaper. So the products that start as cheap, simple options tend to gradually grow more powerful till, like water rising in a room, they squash the "high-end" products against the ceiling. Sun did this to mainframes, and Intel is doing it to Sun. Microsoft Word did it to desktop publishing software like Interleaf and Framemaker. Mass-market digital cameras are doing it to the expensive models made for professionals. Avid did it to the manufacturers of specialized video editing systems, and now Apple is doing it to Avid. *Henry Ford* did it to the car makers that preceded him. If you build the simple, inexpensive option, you'll not only find it easier to sell at first, but you'll also be in the best position to conquer the rest of the market.

It's very dangerous to let anyone fly under you. If you have the cheapest, easiest

product, you'll own the low end. And if you don't, you're in the crosshairs of whoever does.

Raising Money

To make all this happen, you're going to need money. Some startups have been self-funding-- Microsoft for example-- but most aren't. I think it's wise to take money from investors. To be self-funding, you have to start as a consulting company, and it's hard to switch from that to a product company.

Financially, a startup is like a pass/fail course. The way to get rich from a startup is to maximize the company's chances of succeeding, not to maximize the amount of stock you retain. So if you can trade stock for something that improves your odds, it's probably a smart move.

To most hackers, getting investors seems like a terrifying and mysterious process. Actually it's merely tedious. I'll try to give an outline of how it works.

The first thing you'll need is a few tens of thousands of dollars to pay your expenses while you develop a prototype. This is called seed capital. Because so little money is involved, raising seed capital is comparatively easy-- at least in the sense of getting a quick yes or no.

Usually you get seed money from individual rich people called "angels." Often they're people who themselves got rich from technology. At the seed stage, investors don't expect you to have an elaborate business plan. Most know that they're supposed to decide quickly. It's not unusual to get a check within a week based on a half-page agreement.

We started Viaweb with $10,000 of seed money from our friend Julian. But he gave us a lot more than money. He's a former CEO and also a corporate lawyer, so he gave us a lot of valuable advice about business, and also did all the legal work of getting us set up as a company. Plus he introduced us to one of the two angel investors who supplied our next round of funding.

Some angels, especially those with technology backgrounds, may be satisfied with a demo and a verbal description of what you plan to do. But many will want a copy of your business plan, if only to remind themselves what they invested in.

Our angels asked for one, and looking back, I'm amazed how much worry it caused me. "Business plan" has that word "business" in it, so I figured it had to be something I'd have to read a book about business plans to write. Well, it doesn't. At this stage, all most investors expect is a brief description of what you plan to do and how you're going to make money from it, and the resumes of the founders. If you just sit down and write out what you've been saying to one another, that should be fine. It shouldn't take more than a couple hours, and you'll probably find that writing it all down gives you more ideas about what to do.

For the angel to have someone to make the check out to, you're going to have to have some kind of company. Merely incorporating yourselves isn't hard. The problem is, for the company to exist, you have to decide who the founders are, and how much stock they each have. If there are two founders with the same qualifications who are both equally committed to the business, that's easy. But if you have a number of people who are expected to contribute in varying degrees, arranging the proportions of stock can be hard. And once you've done it, it tends to be set in stone.

I have no tricks for dealing with this problem. All I can say is, try hard to do it right. I do have a rule of thumb for recognizing when you have, though. When everyone feels they're getting a slightly bad deal, that they're doing more than they should for the amount of stock they have, the stock is optimally apportioned.

There is more to setting up a company than incorporating it, of course: insurance, business license, unemployment compensation, various things with the IRS. I'm not even sure what the list is, because we, ah, skipped all that. When we got real funding near the end of 1996, we hired a great CFO, who fixed everything retroactively. It turns out that no one comes and arrests you if you don't do everything you're supposed to when starting a company. And a good thing too, or a lot of startups would never get started. [5]

It can be dangerous to delay turning yourself into a company, because one or more of the founders might decide to split off and start another company doing the same thing. This does happen. So when you set up the company, as well as as apportioning the stock, you should get all the founders to sign something agreeing that everyone's ideas belong to this company, and that this company is going to be everyone's only job.

[If this were a movie, ominous music would begin here.]

While you're at it, you should ask what else they've signed. One of the worst things that can happen to a startup is to run into intellectual property problems. We did, and it came closer to killing us than any competitor ever did.

As we were in the middle of getting bought, we discovered that one of our people had, early on, been bound by an agreement that said all his ideas belonged to the giant company that was paying for him to go to grad school. In theory, that could have meant someone else owned big chunks of our software. So the acquisition came to a screeching halt while we tried to sort this out. The problem was, since we'd been about to be acquired, we'd allowed ourselves to run low on cash. Now we needed to raise more to keep going. But it's hard to raise money with an IP cloud over your head, because investors can't judge how serious it is.

Our existing investors, knowing that we needed money and had nowhere else to get it, at this point attempted certain gambits which I will not describe in detail, except to remind readers that the word "angel" is a metaphor. The founders thereupon proposed to walk away from the company, after giving the investors a

brief tutorial on how to administer the servers themselves. And while this was happening, the acquirers used the delay as an excuse to welch on the deal.

Miraculously it all turned out ok. The investors backed down; we did another round of funding at a reasonable valuation; the giant company finally gave us a piece of paper saying they didn't own our software; and six months later we were bought by Yahoo for much more than the earlier acquirer had agreed to pay. So we were happy in the end, though the experience probably took several years off my life.

Don't do what we did. Before you consummate a startup, ask everyone about their previous IP history.

Once you've got a company set up, it may seem presumptuous to go knocking on the doors of rich people and asking them to invest tens of thousands of dollars in something that is really just a bunch of guys with some ideas. But when you look at it from the rich people's point of view, the picture is more encouraging. Most rich people are looking for good investments. If you really think you have a chance of succeeding, you're doing them a favor by letting them invest. Mixed with any annoyance they might feel about being approached will be the thought: are these guys the next Google?

Usually angels are financially equivalent to founders. They get the same kind of stock and get diluted the same amount in future rounds. How much stock should they get? That depends on how ambitious you feel. When you offer x percent of your company for y dollars, you're implicitly claiming a certain value for the whole company. Venture investments are usually described in terms of that number. If you give an investor new shares equal to 5% of those already outstanding in return for $100,000, then you've done the deal at a pre-money valuation of $2 million.

How do you decide what the value of the company should be? There is no rational way. At this stage the company is just a bet. I didn't realize that when we were raising money. Julian thought we ought to value the company at several million dollars. I thought it was preposterous to claim that a couple thousand lines of code, which was all we had at the time, were worth several million dollars. Eventually we settled on one million, because Julian said no one would invest in a company with a valuation any lower. [6]

What I didn't grasp at the time was that the valuation wasn't just the value of the code we'd written so far. It was also the value of our ideas, which turned out to be right, and of all the future work we'd do, which turned out to be a lot.

The next round of funding is the one in which you might deal with actual venture capital firms. But don't wait till you've burned through your last round of funding to start approaching them. VCs are slow to make up their minds. They can take months. You don't want to be running out of money while you're trying to negotiate with them.

Getting money from an actual VC firm is a bigger deal than getting money from angels. The amounts of money involved are larger, millions usually. So the deals take longer, dilute you more, and impose more onerous conditions.

Sometimes the VCs want to install a new CEO of their own choosing. Usually the claim is that you need someone mature and experienced, with a business background. Maybe in some cases this is true. And yet Bill Gates was young and inexperienced and had no business background, and he seems to have done ok. Steve Jobs got booted out of his own company by someone mature and experienced, with a business background, who then proceeded to ruin the company. So I think people who are mature and experienced, with a business background, may be overrated. We used to call these guys "newscasters," because they had neat hair and spoke in deep, confident voices, and generally didn't know much more than they read on the teleprompter.

We talked to a number of VCs, but eventually we ended up financing our startup entirely with angel money. The main reason was that we feared a brand-name VC firm would stick us with a newscaster as part of the deal. That might have been ok if he was content to limit himself to talking to the press, but what if he wanted to have a say in running the company? That would have led to disaster, because our software was so complex. We were a company whose whole m.o. was to win through better technology. The strategic decisions were mostly decisions about technology, and we didn't need any help with those.

This was also one reason we didn't go public. Back in 1998 our CFO tried to talk me into it. In those days you could go public as a dogfood portal, so as a company with a real product and real revenues, we might have done well. But I feared it would have meant taking on a newscaster-- someone who, as they say, "can talk Wall Street's language."

I'm happy to see Google is bucking that trend. They didn't talk Wall Street's language when they did their IPO, and Wall Street didn't buy. And now Wall Street is collectively kicking itself. They'll pay attention next time. Wall Street learns new languages fast when money is involved.

You have more leverage negotiating with VCs than you realize. The reason is other VCs. I know a number of VCs now, and when you talk to them you realize that it's a seller's market. Even now there is too much money chasing too few good deals.

VCs form a pyramid. At the top are famous ones like Sequoia and Kleiner Perkins, but beneath those are a huge number you've never heard of. What they all have in common is that a dollar from them is worth one dollar. Most VCs will tell you that they don't just provide money, but connections and advice. If you're talking to Vinod Khosla or John Doerr or Mike Moritz, this is true. But such advice and connections can come very expensive. And as you go down the food chain the VCs get rapidly dumber. A few steps down from the top you're basically talking to bankers who've picked up a few new vocabulary words from reading *Wired*. (Does your product use *XML?*) So I'd advise you to be skeptical about claims of

experience and connections. Basically, a VC is a source of money. I'd be inclined to go with whoever offered the most money the soonest with the least strings attached.

You may wonder how much to tell VCs. And you should, because some of them may one day be funding your competitors. I think the best plan is not to be overtly secretive, but not to tell them everything either. After all, as most VCs say, they're more interested in the people than the ideas. The main reason they want to talk about your idea is to judge you, not the idea. So as long as you seem like you know what you're doing, you can probably keep a few things back from them. [7]

Talk to as many VCs as you can, even if you don't want their money, because a) they may be on the board of someone who will buy you, and b) if you seem impressive, they'll be discouraged from investing in your competitors. The most efficient way to reach VCs, especially if you only want them to know about you and don't want their money, is at the conferences that are occasionally organized for startups to present to them.

Not Spending It

When and if you get an infusion of real money from investors, what should you do with it? Not spend it, that's what. In nearly every startup that fails, the proximate cause is running out of money. Usually there is something deeper wrong. But even a proximate cause of death is worth trying hard to avoid.

During the Bubble many startups tried to "get big fast." Ideally this meant getting a lot of customers fast. But it was easy for the meaning to slide over into hiring a lot of people fast.

Of the two versions, the one where you get a lot of customers fast is of course preferable. But even that may be overrated. The idea is to get there first and get all the users, leaving none for competitors. But I think in most businesses the advantages of being first to market are not so overwhelmingly great. Google is again a case in point. When they appeared it seemed as if search was a mature market, dominated by big players who'd spent millions to build their brands: Yahoo, Lycos, Excite, Infoseek, Altavista, Inktomi. Surely 1998 was a little late to arrive at the party.

But as the founders of Google knew, brand is worth next to nothing in the search business. You can come along at any point and make something better, and users will gradually seep over to you. As if to emphasize the point, Google never did any advertising. They're like dealers; they sell the stuff, but they know better than to use it themselves.

The competitors Google buried would have done better to spend those millions improving their software. Future startups should learn from that mistake. Unless you're in a market where products are as undifferentiated as cigarettes or vodka or laundry detergent, spending a lot on brand advertising is a sign of breakage. And

few if any Web businesses are so undifferentiated. The dating sites are running big ad campaigns right now, which is all the more evidence they're ripe for the picking. (Fee, fie, fo, fum, I smell a company run by marketing guys.)

We were compelled by circumstances to grow slowly, and in retrospect it was a good thing. The founders all learned to do every job in the company. As well as writing software, I had to do sales and customer support. At sales I was not very good. I was persistent, but I didn't have the smoothness of a good salesman. My message to potential customers was: you'd be stupid not to sell online, and if you sell online you'd be stupid to use anyone else's software. Both statements were true, but that's not the way to convince people.

I was great at customer support though. Imagine talking to a customer support person who not only knew everything about the product, but would apologize abjectly if there was a bug, and then fix it immediately, while you were on the phone with them. Customers loved us. And we loved them, because when you're growing slow by word of mouth, your first batch of users are the ones who were smart enough to find you by themselves. There is nothing more valuable, in the early stages of a startup, than smart users. If you listen to them, they'll tell you exactly how to make a winning product. And not only will they give you this advice for free, they'll pay you.

We officially launched in early 1996. By the end of that year we had about 70 users. Since this was the era of "get big fast," I worried about how small and obscure we were. But in fact we were doing exactly the right thing. Once you get big (in users or employees) it gets hard to change your product. That year was effectively a laboratory for improving our software. By the end of it, we were so far ahead of our competitors that they never had a hope of catching up. And since all the hackers had spent many hours talking to users, we understood online commerce way better than anyone else.

That's the key to success as a startup. There is nothing more important than understanding your business. You might think that anyone in a business must, ex officio, understand it. Far from it. Google's secret weapon was simply that they understood search. I was working for Yahoo when Google appeared, and Yahoo didn't understand search. I know because I once tried to convince the powers that be that we had to make search better, and I got in reply what was then the party line about it: that Yahoo was no longer a mere "search engine." Search was now only a small percentage of our page views, less than one month's growth, and now that we were established as a "media company," or "portal," or whatever we were, search could safely be allowed to wither and drop off, like an umbilical cord.

Well, a small fraction of page views they may be, but they are an important fraction, because they are the page views that Web sessions start with. I think Yahoo gets that now.

Google understands a few other things most Web companies still don't. The most important is that you should put users before advertisers, even though the

advertisers are paying and users aren't. One of my favorite bumper stickers reads "if the people lead, the leaders will follow." Paraphrased for the Web, this becomes "get all the users, and the advertisers will follow." More generally, design your product to please users first, and then think about how to make money from it. If you don't put users first, you leave a gap for competitors who do.

To make something users love, you have to understand them. And the bigger you are, the harder that is. So I say "get big slow." The slower you burn through your funding, the more time you have to learn.

The other reason to spend money slowly is to encourage a culture of cheapness. That's something Yahoo did understand. David Filo's title was "Chief Yahoo," but he was proud that his unofficial title was "Cheap Yahoo." Soon after we arrived at Yahoo, we got an email from Filo, who had been crawling around our directory hierarchy, asking if it was really necessary to store so much of our data on expensive RAID drives. I was impressed by that. Yahoo's market cap then was already in the billions, and they were still worrying about wasting a few gigs of disk space.

When you get a couple million dollars from a VC firm, you tend to feel rich. It's important to realize you're not. A rich company is one with large revenues. This money isn't revenue. It's money investors have given you in the hope you'll be able to generate revenues. So despite those millions in the bank, you're still poor.

For most startups the model should be grad student, not law firm. Aim for cool and cheap, not expensive and impressive. For us the test of whether a startup understood this was whether they had Aeron chairs. The Aeron came out during the Bubble and was very popular with startups. Especially the type, all too common then, that was like a bunch of kids playing house with money supplied by VCs. We had office chairs so cheap that the arms all fell off. This was slightly embarrassing at the time, but in retrospect the grad-studenty atmosphere of our office was another of those things we did right without knowing it.

Our offices were in a wooden triple-decker in Harvard Square. It had been an apartment until about the 1970s, and there was still a claw-footed bathtub in the bathroom. It must once have been inhabited by someone fairly eccentric, because a lot of the chinks in the walls were stuffed with aluminum foil, as if to protect against cosmic rays. When eminent visitors came to see us, we were a bit sheepish about the low production values. But in fact that place was the perfect space for a startup. We felt like our role was to be impudent underdogs instead of corporate stuffed shirts, and that is exactly the spirit you want.

An apartment is also the right kind of place for developing software. Cube farms suck for that, as you've probably discovered if you've tried it. Ever notice how much easier it is to hack at home than at work? So why not make work more like home?

When you're looking for space for a startup, don't feel that it has to look

professional. Professional means doing good work, not elevators and glass walls. I'd advise most startups to avoid corporate space at first and just rent an apartment. You want to live at the office in a startup, so why not have a place designed to be lived in as your office?

Besides being cheaper and better to work in, apartments tend to be in better locations than office buildings. And for a startup location is very important. The key to productivity is for people to come back to work after dinner. Those hours after the phone stops ringing are by far the best for getting work done. Great things happen when a group of employees go out to dinner together, talk over ideas, and then come back to their offices to implement them. So you want to be in a place where there are a lot of restaurants around, not some dreary office park that's a wasteland after 6:00 PM. Once a company shifts over into the model where everyone drives home to the suburbs for dinner, however late, you've lost something extraordinarily valuable. God help you if you actually start in that mode.

If I were going to start a startup today, there are only three places I'd consider doing it: on the Red Line near Central, Harvard, or Davis Squares (Kendall is too sterile); in Palo Alto on University or California Aves; and in Berkeley immediately north or south of campus. These are the only places I know that have the right kind of vibe.

The most important way to not spend money is by not hiring people. I may be an extremist, but I think hiring people is the worst thing a company can do. To start with, people are a recurring expense, which is the worst kind. They also tend to cause you to grow out of your space, and perhaps even move to the sort of uncool office building that will make your software worse. But worst of all, they slow you down: instead of sticking your head in someone's office and checking out an idea with them, eight people have to have a meeting about it. So the fewer people you can hire, the better.

During the Bubble a lot of startups had the opposite policy. They wanted to get "staffed up" as soon as possible, as if you couldn't get anything done unless there was someone with the corresponding job title. That's big company thinking. Don't hire people to fill the gaps in some a priori org chart. The only reason to hire someone is to do something you'd like to do but can't.

If hiring unnecessary people is expensive and slows you down, why do nearly all companies do it? I think the main reason is that people like the idea of having a lot of people working for them. This weakness often extends right up to the CEO. If you ever end up running a company, you'll find the most common question people ask is how many employees you have. This is their way of weighing you. It's not just random people who ask this; even reporters do. And they're going to be a lot more impressed if the answer is a thousand than if it's ten.

This is ridiculous, really. If two companies have the same revenues, it's the one with fewer employees that's more impressive. When people used to ask me how many people our startup had, and I answered "twenty," I could see them thinking

that we didn't count for much. I used to want to add "but our main competitor, whose ass we regularly kick, has a hundred and forty, so can we have credit for the larger of the two numbers?"

As with office space, the number of your employees is a choice between seeming impressive, and being impressive. Any of you who were [nerds] in high school know about this choice. Keep doing it when you start a company.

Should You?

But should you start a company? Are you the right sort of person to do it? If you are, is it worth it?

More people are the right sort of person to start a startup than realize it. That's the main reason I wrote this. There could be ten times more startups than there are, and that would probably be a good thing.

I was, I now realize, exactly the right sort of person to start a startup. But the idea terrified me at first. I was forced into it because I was a [Lisp] hacker. The company I'd been consulting for seemed to be running into trouble, and there were not a lot of other companies using Lisp. Since I couldn't bear the thought of programming in another language (this was 1995, remember, when "another language" meant C++) the only option seemed to be to start a new company using Lisp.

I realize this sounds far-fetched, but if you're a Lisp hacker you'll know what I mean. And if the idea of starting a startup frightened me so much that I only did it out of necessity, there must be a lot of people who would be good at it but who are too intimidated to try.

So who should start a startup? Someone who is a good hacker, between about 23 and 38, and who wants to solve the money problem in one shot instead of getting paid gradually over a conventional working life.

I can't say precisely what a good hacker is. At a first rate university this might include the top half of computer science majors. Though of course you don't have to be a CS major to be a hacker; I was a philosophy major in college.

It's hard to tell whether you're a good hacker, especially when you're young. Fortunately the process of starting startups tends to select them automatically. What drives people to start startups is (or should be) looking at existing technology and thinking, don't these guys realize they should be doing x, y, and z? And that's also a sign that one is a good hacker.

I put the lower bound at 23 not because there's something that doesn't happen to your brain till then, but because you need to see what it's like in an existing business before you try running your own. The business doesn't have to be a startup. I spent a year working for a software company to pay off my college loans. It was the worst year of my adult life, but I learned, without realizing it at

the time, a lot of valuable lessons about the software business. In this case they were mostly negative lessons: don't have a lot of meetings; don't have chunks of code that multiple people own; don't have a sales guy running the company; don't make a high-end product; don't let your code get too big; don't leave finding bugs to QA people; don't go too long between releases; don't isolate developers from users; don't move from Cambridge to Route 128; and so on. [8] But negative lessons are just as valuable as positive ones. Perhaps even more valuable: it's hard to repeat a brilliant performance, but it's straightforward to avoid errors. [9]

The other reason it's hard to start a company before 23 is that people won't take you seriously. VCs won't trust you, and will try to reduce you to a mascot as a condition of funding. Customers will worry you're going to flake out and leave them stranded. Even you yourself, unless you're very unusual, will feel your age to some degree; you'll find it awkward to be the boss of someone much older than you, and if you're 21, hiring only people younger rather limits your options.

Some people could probably start a company at 18 if they wanted to. Bill Gates was 19 when he and Paul Allen started Microsoft. (Paul Allen was 22, though, and that probably made a difference.) So if you're thinking, I don't care what he says, I'm going to start a company now, you may be the sort of person who could get away with it.

The other cutoff, 38, has a lot more play in it. One reason I put it there is that I don't think many people have the physical stamina much past that age. I used to work till 2:00 or 3:00 AM every night, seven days a week. I don't know if I could do that now.

Also, startups are a big risk financially. If you try something that blows up and leaves you broke at 26, big deal; a lot of 26 year olds are broke. By 38 you can't take so many risks-- especially if you have kids.

My final test may be the most restrictive. Do you actually want to start a startup? What it amounts to, economically, is compressing your working life into the smallest possible space. Instead of working at an ordinary rate for 40 years, you work like hell for four. And maybe end up with nothing-- though in that case it probably won't take four years.

During this time you'll do little but work, because when you're not working, your competitors will be. My only leisure activities were running, which I needed to do to keep working anyway, and about fifteen minutes of reading a night. I had a girlfriend for a total of two months during that three year period. Every couple weeks I would take a few hours off to visit a used bookshop or go to a friend's house for dinner. I went to visit my family twice. Otherwise I just worked.

Working was often fun, because the people I worked with were some of my best friends. Sometimes it was even technically interesting. But only about 10% of the time. The best I can say for the other 90% is that some of it is funnier in hindsight than it seemed then. Like the time the power went off in Cambridge for about six

hours, and we made the mistake of trying to start a gasoline powered generator inside our offices. I won't try that again.

I don't think the amount of bullshit you have to deal with in a startup is more than you'd endure in an ordinary working life. It's probably less, in fact; it just seems like a lot because it's compressed into a short period. So mainly what a startup buys you is time. That's the way to think about it if you're trying to decide whether to start one. If you're the sort of person who would like to solve the money problem once and for all instead of working for a salary for 40 years, then a startup makes sense.

For a lot of people the conflict is between startups and graduate school. Grad students are just the age, and just the sort of people, to start software startups. You may worry that if you do you'll blow your chances of an academic career. But it's possible to be part of a startup and stay in grad school, especially at first. Two of our three original hackers were in grad school the whole time, and both got their degrees. There are few sources of energy so powerful as a procrastinating grad student.

If you do have to leave grad school, in the worst case it won't be for too long. If a startup fails, it will probably fail quickly enough that you can return to academic life. And if it succeeds, you may find you no longer have such a burning desire to be an assistant professor.

If you want to do it, do it. Starting a startup is not the great mystery it seems from outside. It's not something you have to know about "business" to do. Build something users love, and spend less than you make. How hard is that?

Notes

[1] Google's revenues are about two billion a year, but half comes from ads on other sites.

[2] One advantage startups have over established companies is that there are no discrimination laws about starting businesses. For example, I would be reluctant to start a startup with a woman who had small children, or was likely to have them soon. But you're not allowed to ask prospective employees if they plan to have kids soon. Believe it or not, under current US law, you're not even allowed to discriminate on the basis of intelligence. Whereas when you're starting a company, you can discriminate on any basis you want about who you start it with.

[3] Learning to hack is a lot cheaper than business school, because you can do it

mostly on your own. For the price of a Linux box, a copy of K&R, and a few hours of advice from your neighbor's fifteen year old son, you'll be well on your way.

[4] Corollary: Avoid starting a startup to sell things to the biggest company of all, the government. Yes, there are lots of opportunities to sell them technology. But let someone else start those startups.

[5] A friend who started a company in Germany told me they do care about the paperwork there, and that there's more of it. Which helps explain why there are not more startups in Germany.

[6] At the seed stage our valuation was in principle $100,000, because Julian got 10% of the company. But this is a very misleading number, because the money was the least important of the things Julian gave us.

[7] The same goes for companies that seem to want to acquire you. There will be a few that are only pretending to in order to pick your brains. But you can never tell for sure which these are, so the best approach is to seem entirely open, but to fail to mention a few critical technical secrets.

[8] I was as bad an employee as this place was a company. I apologize to anyone who had to work with me there.

[9] You could probably write a book about how to succeed in business by doing everything in exactly the opposite way from the DMV.

Thanks to Trevor Blackwell, Sarah Harlin, Jessica Livingston, and Robert Morris for reading drafts of this essay, and to Steve Melendez and Gregory Price for inviting me to speak.

·

[Domain Name Search](#)

·

[Turkish Translation](#)

·

[Hebrew Translation](#)

-

[Russian Translation](#)

-

[Chinese Translation](#)

-

[French Translation](#)

-

[Japanese Translation](#)

-

[Arabic Translation](#)

A Unified Theory of VC Suckage

March 2005

A couple months ago I got an email from a recruiter asking if I was interested in being a "technologist in residence" at a new venture capital fund. I think the idea was to play Karl Rove to the VCs' George Bush.

I considered it for about four seconds. Work for a VC fund? Ick.

One of my most vivid memories from our startup is going to visit Greylock, the famous Boston VCs. They were the most arrogant people I've met in my life. And I've met a lot of arrogant people. [1]

I'm not alone in feeling this way, of course. Even a VC friend of mine dislikes VCs. "Assholes," he says.

But lately I've been learning more about how the VC world works, and a few days ago it hit me that there's a reason VCs are the way they are. It's not so much that the business attracts jerks, or even that the power they wield corrupts them. The real problem is the way they're paid.

The problem with VC funds is that they're *funds*. Like the managers of mutual funds or hedge funds, VCs get paid a percentage of the money they manage: about 2% a year in management fees, plus a percentage of the gains. So they want the fund to be huge-- hundreds of millions of dollars, if possible. But that means each partner ends up being responsible for investing a lot of money. And since one person can only manage so many deals, each deal has to be for multiple millions of dollars.

This turns out to explain nearly all the characteristics of VCs that founders hate.

It explains why VCs take so agonizingly long to make up their minds, and why their due diligence feels like a body cavity search. [2] With so much at stake, they have to be paranoid.

It explains why they steal your ideas. Every founder knows that VCs will tell your secrets to your competitors if they end up investing in them. It's not unheard of for VCs to meet you when they have no intention of funding you, just to pick your brain for a competitor. This prospect makes naive founders clumsily secretive.

Experienced founders treat it as a cost of doing business. Either way it sucks. But again, the only reason VCs are so sneaky is the giant deals they do. With so much at stake, they have to be devious.

It explains why VCs tend to interfere in the companies they invest in. They want to be on your board not just so that they can advise you, but so that they can watch you. Often they even install a new CEO. Yes, he may have extensive business experience. But he's also their man: these newly installed CEOs always play something of the role of a political commissar in a Red Army unit. With so much at stake, VCs can't resist micromanaging you.

The huge investments themselves are something founders would dislike, if they realized how damaging they can be. VCs don't invest $x million because that's the amount you need, but because that's the amount the structure of their business requires them to invest. Like steroids, these sudden huge investments can do more harm than good. Google survived enormous VC funding because it could legitimately absorb large amounts of money. They had to buy a lot of servers and a lot of bandwidth to crawl the whole Web. Less fortunate startups just end up hiring armies of people to sit around having meetings.

In principle you could take a huge VC investment, put it in treasury bills, and continue to operate frugally. You just try it.

And of course giant investments mean giant valuations. They have to, or there's not enough stock left to keep the founders interested. You might think a high valuation is a great thing. Many founders do. But you can't eat paper. You can't benefit from a high valuation unless you can somehow achieve what those in the business call a "liquidity event," and the higher your valuation, the narrower your options for doing that. Many a founder would be happy to sell his company for $15 million, but VCs who've just invested at a pre-money valuation of $8 million won't hear of that. You're rolling the dice again, whether you like it or not.

Back in 1997, one of our competitors raised $20 million in a single round of VC funding. This was at the time more than the valuation of our entire company. Was I worried? Not at all: I was delighted. It was like watching a car you're chasing turn down a street that you know has no outlet.

Their smartest move at that point would have been to take every penny of the $20 million and use it to buy us. We would have sold. Their investors would have been furious of course. But I think the main reason they never considered this was that they never imagined we could be had so cheap. They probably assumed we were on the same VC gravy train they were.

In fact we only spent about $2 million in our entire existence. And that gave us flexibility. We could sell ourselves to Yahoo for $50 million, and everyone was delighted. If our competitor had done that, the last round of investors would presumably have lost money. I assume they could have vetoed such a deal. But no one those days was paying a lot more than Yahoo. So unless their founders could

pull off an IPO (which would be difficult with Yahoo as a competitor), they had no choice but to ride the thing down.

The puffed-up companies that went public during the Bubble didn't do it just because they were pulled into it by unscrupulous investment bankers. Most were pushed just as hard from the other side by VCs who'd invested at high valuations, leaving an IPO as the only way out. The only people dumber were retail investors. So it was literally IPO or bust. Or rather, IPO then bust, or just bust.

Add up all the evidence of VCs' behavior, and the resulting personality is not attractive. In fact, it's the classic villain: alternately cowardly, greedy, sneaky, and overbearing.

I used to take it for granted that VCs were like this. Complaining that VCs were jerks used to seem as naive to me as complaining that users didn't read the reference manual. Of course VCs were jerks. How could it be otherwise?

But I realize now that they're not intrinsically jerks. VCs are like car salesmen or bureaucrats: the nature of their work turns them into jerks.

I've met a few VCs I like. Mike Moritz seems a good guy. He even has a sense of humor, which is almost unheard of among VCs. From what I've read about John Doerr, he sounds like a good guy too, almost a hacker. But they work for the very best VC funds. And my theory explains why they'd tend to be different: just as the very most popular kids don't have to persecute nerds, the very best VCs don't have to act like VCs. They get the pick of all the best deals. So they don't have to be so paranoid and sneaky, and they can choose those rare companies, like Google, that will actually benefit from the giant sums they're compelled to invest.

VCs often complain that in their business there's too much money chasing too few deals. Few realize that this also describes a flaw in the way funding works at the level of individual firms.

Perhaps this was the sort of strategic insight I was supposed to come up with as a "technologist in residence." If so, the good news is that they're getting it for free. The bad news is it means that if you're not one of the very top funds, you're condemned to be the bad guys.

Notes

[1] After Greylock booted founder Philip Greenspun out of ArsDigita, he wrote a hilarious but also very informative essay about it.

[2] Since most VCs aren't tech guys, the technology side of their due diligence tends to be like a body cavity search by someone with a faulty knowledge of human anatomy. After a while we were quite sore from VCs attempting to probe

our nonexistent database orifice.

No, we don't use Oracle. We just store the data in files. Our secret is to use an OS that doesn't lose our data. Which OS? FreeBSD. Why do you use that instead of Windows NT? Because it's better and it doesn't cost anything. What, you're using a *freeware* OS?

How many times that conversation was repeated. Then when we got to Yahoo, we found they used FreeBSD and stored their data in files too.

■

[Chinese Translation](#)

■

[Japanese Translation](#)

Undergraduation

March 2005

(Parts of this essay began as replies to students who wrote to me with questions.)

Recently I've had several emails from computer science undergrads asking what to do in college. I might not be the best source of advice, because I was a philosophy major in college. But I took so many CS classes that most CS majors thought I was one. I was certainly a hacker, at least.

Hacking

What should you do in college to become a [good hacker](#)? There are two main things you can do: become very good at programming, and learn a lot about specific, cool problems. These turn out to be equivalent, because each drives you to do the other.

The way to be good at programming is to work (a) a lot (b) on hard problems. And the way to make yourself work on hard problems is to work on some very engaging project.

Odds are this project won't be a class assignment. My friend Robert learned a lot by writing network software when he was an undergrad. One of his projects was to connect Harvard to the Arpanet; it had been one of the original nodes, but by 1984 the connection had died. [1] Not only was this work not for a class, but because he spent all his time on it and neglected his studies, he was kicked out of school for a year. [2] It all evened out in the end, and now he's a professor at MIT. But you'll probably be happier if you don't go to that extreme; it caused him a lot of worry at the time.

Another way to be good at programming is to find other people who are good at it,

and learn what they know. Programmers tend to sort themselves into tribes according to the type of work they do and the tools they use, and some tribes are smarter than others. Look around you and see what the smart people seem to be working on; there's usually a reason.

Some of the smartest people around you are professors. So one way to find interesting work is to volunteer as a research assistant. Professors are especially interested in people who can solve tedious system-administration type problems for them, so that is a way to get a foot in the door. What they fear are flakes and resume padders. It's all too common for an assistant to result in a net increase in work. So you have to make it clear you'll mean a net decrease.

Don't be put off if they say no. Rejection is almost always less personal than the rejectee imagines. Just move on to the next. (This applies to dating too.)

Beware, because although most professors are smart, not all of them work on interesting stuff. Professors have to publish novel results to advance their careers, but there is more competition in more interesting areas of research. So what less ambitious professors do is turn out a series of papers whose conclusions are novel because no one else cares about them. You're better off avoiding these.

I never worked as a research assistant, so I feel a bit dishonest recommending that route. I learned to program by writing stuff of my own, particularly by trying to reverse-engineer Winograd's SHRDLU. I was as obsessed with that program as a mother with a new baby.

Whatever the disadvantages of working by yourself, the advantage is that the project is all your own. You never have to compromise or ask anyone's permission, and if you have a new idea you can just sit down and start implementing it.

In your own projects you don't have to worry about novelty (as professors do) or profitability (as businesses do). All that matters is how hard the project is technically, and that has no correlation to the nature of the application. "Serious" applications like databases are often trivial and dull technically (if you ever suffer from insomnia, try reading the technical literature about databases) while "frivolous" applications like games are often very sophisticated. I'm sure there are game companies out there working on products with more intellectual content than the research at the bottom nine tenths of university CS departments.

If I were in college now I'd probably work on graphics: a network game, for example, or a tool for 3D animation. When I was an undergrad there weren't enough cycles around to make graphics interesting, but it's hard to imagine anything more fun to work on now.

Math

When I was in college, a lot of the professors believed (or at least wished) that computer science was a branch of math. This idea was strongest at Harvard, where

there wasn't even a CS major till the 1980s; till then one had to major in applied math. But it was nearly as bad at Cornell. When I told the fearsome Professor Conway that I was interested in AI (a hot topic then), he told me I should major in math. I'm still not sure whether he thought AI required math, or whether he thought AI was nonsense and that majoring in something rigorous would cure me of such stupid ambitions.

In fact, the amount of math you need as a hacker is a lot less than most university departments like to admit. I don't think you need much more than high school math plus a few concepts from the theory of computation. (You have to know what an n^2 algorithm is if you want to avoid writing them.) Unless you're planning to write math applications, of course. Robotics, for example, is all math.

But while you don't literally need math for most kinds of hacking, in the sense of knowing 1001 tricks for differentiating formulas, math is very much worth studying for its own sake. It's a valuable source of metaphors for almost any kind of work. [3] I wish I'd studied more math in college for that reason.

Like a lot of people, I was mathematically abused as a child. I learned to think of math as a collection of formulas that were neither beautiful nor had any relation to my life (despite attempts to translate them into "word problems"), but had to be memorized in order to do well on tests.

One of the most valuable things you could do in college would be to learn what math is really about. This may not be easy, because a lot of good mathematicians are bad teachers. And while there are many popular books on math, few seem good. The best I can think of are W. W. Sawyer's. And of course Euclid. [4]

Everything

Thomas Huxley said "Try to learn something about everything and everything about something." Most universities aim at this ideal.

But what's everything? To me it means, all that people learn in the course of working honestly on hard problems. All such work tends to be related, in that ideas and techniques from one field can often be transplanted successfully to others. Even others that seem quite distant. For example, I write essays the same way I write software: I sit down and blow out a lame version 1 as fast as I can type, then spend several weeks rewriting it.

Working on hard problems is not, by itself, enough. Medieval alchemists were working on a hard problem, but their approach was so bogus that there was little to learn from studying it, except possibly about people's ability to delude themselves. Unfortunately the sort of AI I was trying to learn in college had the same flaw: a very hard problem, blithely approached with hopelessly inadequate techniques. Bold? Closer to fraudulent.

The social sciences are also fairly bogus, because they're so much influenced by

intellectual [fashions](). If a physicist met a colleague from 100 years ago, he could teach him some new things; if a psychologist met a colleague from 100 years ago, they'd just get into an ideological argument. Yes, of course, you'll learn something by taking a psychology class. The point is, you'll learn more by taking a class in another department.

The worthwhile departments, in my opinion, are math, the hard sciences, engineering, history (especially economic and social history, and the history of science), architecture, and the classics. A survey course in art history may be worthwhile. Modern literature is important, but the way to learn about it is just to read. I don't know enough about music to say.

You can skip the social sciences, philosophy, and the various departments created recently in response to political pressures. Many of these fields talk about important problems, certainly. But the way they talk about them is useless. For example, philosophy talks, among other things, about our obligations to one another; but you can learn more about this from a wise grandmother or E. B. White than from an academic philosopher.

I speak here from experience. I should probably have been offended when people laughed at Clinton for saying "It depends on what the meaning of the word 'is' is." I took about five classes in college on what the meaning of "is" is.

Another way to figure out which fields are worth studying is to create the *dropout graph.* For example, I know many people who switched from math to computer science because they found math too hard, and no one who did the opposite. People don't do hard things gratuitously; no one will work on a harder problem unless it is proportionately (or at least log(n)) more rewarding. So probably math is more worth studying than computer science. By similar comparisons you can make a graph of all the departments in a university. At the bottom you'll find the subjects with least intellectual content.

If you use this method, you'll get roughly the same answer I just gave.

Language courses are an anomaly. I think they're better considered as extracurricular activities, like pottery classes. They'd be far more useful when combined with some time living in a country where the language is spoken. On a whim I studied Arabic as a freshman. It was a lot of work, and the only lasting benefits were a weird ability to identify semitic roots and some insights into how people recognize words.

Studio art and creative writing courses are wildcards. Usually you don't get taught much: you just work (or don't work) on whatever you want, and then sit around offering "crits" of one another's creations under the vague supervision of the teacher. But writing and art are both very hard problems that (some) people work honestly at, so they're worth doing, especially if you can find a good teacher.

Jobs

Of course college students have to think about more than just learning. There are also two practical problems to consider: jobs, and graduate school.

In theory a liberal education is not supposed to supply job training. But everyone knows this is a bit of a fib. Hackers at every college learn practical skills, and not by accident.

What you should learn to get a job depends on the kind you want. If you want to work in a big company, learn how to hack Blub on Windows. If you want to work at a cool little company or research lab, you'll do better to learn Ruby on Linux. And if you want to start your own company, which I think will be more and more common, master the most powerful tools you can find, because you're going to be in a race against your competitors, and they'll be your horse.

There is not a direct correlation between the skills you should learn in college and those you'll use in a job. You should aim slightly high in college.

In workouts a football player may bench press 300 pounds, even though he may never have to exert anything like that much force in the course of a game. Likewise, if your professors try to make you learn stuff that's more advanced than you'll need in a job, it may not just be because they're academics, detached from the real world. They may be trying to make you lift weights with your brain.

The programs you write in classes differ in three critical ways from the ones you'll write in the real world: they're small; you get to start from scratch; and the problem is usually artificial and predetermined. In the real world, programs are bigger, tend to involve existing code, and often require you to figure out what the problem is before you can solve it.

You don't have to wait to leave (or even enter) college to learn these skills. If you want to learn how to deal with existing code, for example, you can contribute to open-source projects. The sort of employer you want to work for will be as impressed by that as good grades on class assignments.

In existing open-source projects you don't get much practice at the third skill, deciding what problems to solve. But there's nothing to stop you starting new projects of your own. And good employers will be even more impressed with that.

What sort of problem should you try to solve? One way to answer that is to ask what you need as a user. For example, I stumbled on a good algorithm for spam filtering because I wanted to stop getting spam. Now what I wish I had was a mail reader that somehow prevented my inbox from filling up. I tend to use my inbox as a todo list. But that's like using a screwdriver to open bottles; what one really wants is a bottle opener.

Grad School

What about grad school? Should you go? And how do you get into a good one?

In principle, grad school is professional training in research, and you shouldn't go unless you want to do research as a career. And yet half the people who get PhDs in CS don't go into research. I didn't go to grad school to become a professor. I went because I wanted to learn more.

So if you're mainly interested in hacking and you go to grad school, you'll find a lot of other people who are similarly out of their element. And if half the people around you are out of their element in the same way you are, are you really out of your element?

There's a fundamental problem in "computer science," and it surfaces in situations like this. No one is sure what "research" is supposed to be. A lot of research is hacking that had to be crammed into the form of an academic paper to yield one more quantum of publication.

So it's kind of misleading to ask whether you'll be at home in grad school, because very few people are quite at home in computer science. The whole field is uncomfortable in its own skin. So the fact that you're mainly interested in hacking shouldn't deter you from going to grad school. Just be warned you'll have to do a lot of stuff you don't like.

Number one will be your dissertation. Almost everyone hates their dissertation by the time they're done with it. The process inherently tends to produce an unpleasant result, like a cake made out of whole wheat flour and baked for twelve hours. Few dissertations are read with pleasure, especially by their authors.

But thousands before you have suffered through writing a dissertation. And aside from that, grad school is close to paradise. Many people remember it as the happiest time of their lives. And nearly all the rest, including me, remember it as a period that would have been, if they hadn't had to write a dissertation. [5]

The danger with grad school is that you don't see the scary part upfront. PhD programs start out as college part 2, with several years of classes. So by the time you face the horror of writing a dissertation, you're already several years in. If you quit now, you'll be a grad-school dropout, and you probably won't like that idea. When Robert got kicked out of grad school for writing the Internet worm of 1988, I envied him enormously for finding a way out without the stigma of failure.

On the whole, grad school is probably better than most alternatives. You meet a lot of smart people, and your glum procrastination will at least be a powerful common bond. And of course you have a PhD at the end. I forgot about that. I suppose that's worth something.

The greatest advantage of a PhD (besides being the union card of academia, of course) may be that it gives you some baseline confidence. For example, the Honeywell thermostats in my house have the most atrocious UI. My mother, who

has the same model, diligently spent a day reading the user's manual to learn how to operate hers. She assumed the problem was with her. But I can think to myself "If someone with a PhD in computer science can't understand this thermostat, it *must* be badly designed."

If you still want to go to grad school after this equivocal recommendation, I can give you solid advice about how to get in. A lot of my friends are CS professors now, so I have the inside story about admissions. It's quite different from college. At most colleges, admissions officers decide who gets in. For PhD programs, the professors do. And they try to do it well, because the people they admit are going to be working for them.

Apparently only recommendations really matter at the best schools. Standardized tests count for nothing, and grades for little. The essay is mostly an opportunity to disqualify yourself by saying something stupid. The only thing professors trust is recommendations, preferably from people they know. [6]

So if you want to get into a PhD program, the key is to impress your professors. And from my friends who are professors I know what impresses them: not merely trying to impress them. They're not impressed by students who get good grades or want to be their research assistants so they can get into grad school. They're impressed by students who get good grades and want to be their research assistants because they're genuinely interested in the topic.

So the best thing you can do in college, whether you want to get into grad school or just be good at hacking, is figure out what you truly like. It's hard to trick professors into letting you into grad school, and impossible to trick problems into letting you solve them. College is where faking stops working. From this point, unless you want to go work for a big company, which is like reverting to high school, the only way forward is through doing what you love.

Notes

[1] No one seems to have minded, which shows how unimportant the Arpanet (which became the Internet) was as late as 1984.

[2] This is why, when I became an employer, I didn't care about GPAs. In fact, we actively sought out people who'd failed out of school. We once put up posters around Harvard saying "Did you just get kicked out for doing badly in your classes because you spent all your time working on some project of your own? Come work for us!" We managed to find a kid who had been, and he was a great hacker.

When Harvard kicks undergrads out for a year, they have to get jobs. The idea is to show them how awful the real world is, so they'll understand how lucky they are to

be in college. This plan backfired with the guy who came to work for us, because he had more fun than he'd had in school, and made more that year from stock options than any of his professors did in salary. So instead of crawling back repentant at the end of the year, he took another year off and went to Europe. He did eventually graduate at about 26.

[3] Eric Raymond says the best metaphors for hackers are in set theory, combinatorics, and graph theory.

Trevor Blackwell reminds you to take math classes intended for math majors. "'Math for engineers' classes sucked mightily. In fact any 'x for engineers' sucks, where x includes math, law, writing and visual design."

[4] Other highly recommended books: *What is Mathematics?*, by Courant and Robbins; *Geometry and the Imagination* by Hilbert and Cohn-Vossen. And for those interested in graphic design, [Byrne's Euclid](#).

[5] If you wanted to have the perfect life, the thing to do would be to go to grad school, secretly write your dissertation in the first year or two, and then just enjoy yourself for the next three years, dribbling out a chapter at a time. This prospect will make grad students' mouths water, but I know of no one who's had the discipline to pull it off.

[6] One professor friend says that 15-20% of the grad students they admit each year are "long shots." But what he means by long shots are people whose applications are perfect in every way, except that no one on the admissions committee knows the professors who wrote the recommendations.

So if you want to get into grad school in the sciences, you need to go to college somewhere with real research professors. Otherwise you'll seem a risky bet to admissions committees, no matter how good you are.

Which implies a surprising but apparently inevitable consequence: little liberal arts colleges are doomed. Most smart high school kids at least consider going into the sciences, even if they ultimately choose not to. Why go to a college that limits their options?

Thanks to Trevor Blackwell, Alex Lewin, Jessica Livingston, Robert Morris, Eric Raymond, and several [anonymous CS professors](#) for reading drafts of this, and to the students whose questions began it.

[More Advice for Undergrads](#)

.

[Joel Spolsky: Advice for Computer Science College Students](#)

.

[Eric Raymond: How to Become a Hacker](#)

Writing, Briefly

March 2005

(In the process of answering an email, I accidentally wrote a tiny essay about writing. I usually spend weeks on an essay. This one took 67 minutes—23 of writing, and 44 of rewriting.)

I think it's far more important to write well than most people realize. Writing doesn't just communicate ideas; it generates them. If you're bad at writing and don't like to do it, you'll miss out on most of the ideas writing would have generated.

As for how to write well, here's the short version: Write a bad version 1 as fast as you can; rewrite it over and over; cut ~~out~~ everything unnecessary; write in a conversational tone; develop a nose for bad writing, so you can see and fix it in yours; imitate writers you like; if you can't get started, tell someone what you plan to write about, then write down what you said; expect 80% of the ideas in an essay to happen after you start writing it, and 50% of those you start with to be wrong; be confident enough to cut; have friends you trust read your stuff and tell you which bits are confusing or drag; don't (always) make detailed outlines; mull ideas over for a few days before writing; carry a small notebook or scrap paper with you; start writing when you think of the first sentence; if a deadline forces you to start before that, just say the most important sentence first; write about stuff you like; don't try to sound impressive; don't hesitate to change the topic on the fly; use footnotes to contain digressions; use anaphora to knit sentences together; read your essays out loud to see (a) where you stumble over awkward phrases and (b) which bits are boring (the paragraphs you dread reading); try to tell the reader something new and useful; work in fairly big quanta of time; when you restart, begin by rereading what you have so far; when you finish, leave yourself something easy to start with; accumulate notes for topics you plan to cover at the bottom of the file; don't feel obliged to cover any of them; write for a reader who won't read the essay as carefully as you do, just as pop songs are designed to sound ok on crappy car radios; if you say anything mistaken, fix it immediately; ask friends which sentence you'll regret most; go back and tone down harsh remarks; publish stuff online, because an audience makes you write more, and thus generate more ideas; print out drafts instead of just looking at them on the screen; use simple, germanic words; learn to distinguish surprises from digressions; learn to recognize the approach of an ending, and when one

appears, grab it.

■

[Russian Translation](#)

■

[Japanese Translation](#)

■

[Romanian Translation](#)

■

[Spanish Translation](#)

■

[German Translation](#)

■

[Chinese Translation](#)

■

[Hungarian Translation](#)

■

[Catalan Translation](#)

[Danish Translation](#)

[Arabic Translation](#)

Return of the Mac

March 2005

All the best hackers I know are gradually switching to Macs. My friend Robert said his whole research group at MIT recently bought themselves Powerbooks. These guys are not the graphic designers and grandmas who were buying Macs at Apple's low point in the mid 1990s. They're about as hardcore OS hackers as you can get.

The reason, of course, is OS X. Powerbooks are beautifully designed and run FreeBSD. What more do you need to know?

I got a Powerbook at the end of last year. When my IBM Thinkpad's hard disk died soon after, it became my only laptop. And when my friend Trevor showed up at my house recently, he was carrying a Powerbook identical to mine.

For most of us, it's not a switch to Apple, but a return. Hard as this was to believe in the mid 90s, the Mac was in its time the canonical hacker's computer.

In the fall of 1983, the professor in one of my college CS classes got up and announced, like a prophet, that there would soon be a computer with half a MIPS of processing power that would fit under an airline seat and cost so little that we could save enough to buy one from a summer job. The whole room gasped. And when the Mac appeared, it was even better than we'd hoped. It was small and powerful and cheap, as promised. But it was also something we'd never considered a computer could be: fabulously well designed.

I had to have one. And I wasn't alone. In the mid to late 1980s, all the hackers I knew were either writing software for the Mac, or wanted to. Every futon sofa in Cambridge seemed to have the same fat white book lying open on it. If you turned it over, it said "Inside Macintosh."

Then came Linux and FreeBSD, and hackers, who follow the most powerful OS wherever it leads, found themselves switching to Intel boxes. If you cared about design, you could buy a Thinkpad, which was at least not actively repellent, if you could get the Intel and Microsoft stickers off the front. [1]

With OS X, the hackers are back. When I walked into the Apple store in Cambridge, it was like coming home. Much was changed, but there was still that Apple coolness in the air, that feeling that the show was being run by someone

who really cared, instead of random corporate deal-makers.

So what, the business world may say. Who cares if hackers like Apple again? How big is the hacker market, after all?

Quite small, but important out of proportion to its size. When it comes to computers, what hackers are doing now, everyone will be doing in ten years. Almost all technology, from Unix to bitmapped displays to the Web, became popular first within CS departments and research labs, and gradually spread to the rest of the world.

I remember telling my father back in 1986 that there was a new kind of computer called a Sun that was a serious Unix machine, but so small and cheap that you could have one of your own to sit in front of, instead of sitting in front of a VT100 connected to a single central Vax. Maybe, I suggested, he should buy some stock in this company. I think he really wishes he'd listened.

In 1994 my friend Koling wanted to talk to his girlfriend in Taiwan, and to save long-distance bills he wrote some software that would convert sound to data packets that could be sent over the Internet. We weren't sure at the time whether this was a proper use of the Internet, which was still then a quasi-government entity. What he was doing is now called VoIP, and it is a huge and rapidly growing business.

If you want to know what ordinary people will be doing with computers in ten years, just walk around the CS department at a good university. Whatever they're doing, you'll be doing.

In the matter of "platforms" this tendency is even more pronounced, because novel software originates with great hackers, and they tend to write it first for whatever computer they personally use. And software sells hardware. Many if not most of the initial sales of the Apple II came from people who bought one to run VisiCalc. And why did Bricklin and Frankston write VisiCalc for the Apple II? Because they personally liked it. They could have chosen any machine to make into a star.

If you want to attract hackers to write software that will sell your hardware, you have to make it something that they themselves use. It's not enough to make it "open." It has to be open and good.

And open and good is what Macs are again, finally. The intervening years have created a situation that is, as far as I know, without precedent: Apple is popular at the low end and the high end, but not in the middle. My seventy year old mother has a Mac laptop. My friends with PhDs in computer science have Mac laptops. [2] And yet Apple's overall market share is still small.

Though unprecedented, I predict this situation is also temporary.

So Dad, there's this company called Apple. They make a new kind of computer that's as well designed as a Bang & Olufsen stereo system, and underneath is the best Unix machine you can buy. Yes, the price to earnings ratio is kind of high, but I think a lot of people are going to want these.

Notes

[1] These horrible stickers are much like the intrusive ads popular on pre-Google search engines. They say to the customer: you are unimportant. We care about Intel and Microsoft, not you.

[2] Y Combinator is (we hope) visited mostly by hackers. The proportions of OSes are: Windows 66.4%, Macintosh 18.8%, Linux 11.4%, and FreeBSD 1.5%. The Mac number is a big change from what it would have been five years ago.

Italian Translation

Russian Translation

Chinese Translation

Why Smart People Have Bad Ideas

April 2005

This summer, as an experiment, some friends and I are giving seed funding to a bunch of new startups. It's an experiment because we're prepared to fund younger founders than most investors would. That's why we're doing it during the summer —so even college students can participate.

We know from Google and Yahoo that grad students can start successful startups. And we know from experience that some undergrads are as capable as most grad students. The accepted age for startup founders has been creeping downward. We're trying to find the lower bound.

The deadline has now passed, and we're sifting through 227 applications. We expected to divide them into two categories, promising and unpromising. But we soon saw we needed a third: promising people with unpromising ideas. [1]

The Artix Phase

We should have expected this. It's very common for a group of founders to go through one lame idea before realizing that a startup has to make something people will pay for. In fact, we ourselves did.

Viaweb wasn't the first startup Robert Morris and I started. In January 1995, we and a couple friends started a company called Artix. The plan was to put art galleries on the Web. In retrospect, I wonder how we could have wasted our time on anything so stupid. Galleries are not especially excited about being on the Web even now, ten years later. They don't want to have their stock visible to any random visitor, like an antique store. [2]

Besides which, art dealers are the most technophobic people on earth. They didn't become art dealers after a difficult choice between that and a career in the hard sciences. Most of them had never seen the Web before we came to tell them why they should be on it. Some didn't even have computers. It doesn't do justice to the situation to describe it as a hard *sell*; we soon sank to building sites for free, and it was hard to convince galleries even to do that.

Gradually it dawned on us that instead of trying to make Web sites for people who didn't want them, we could make sites for people who did. In fact, software that would let people who wanted sites make their own. So we ditched Artix and started a new company, Viaweb, to make software for building online stores. That one succeeded.

We're in good company here. Microsoft was not the first company Paul Allen and Bill Gates started either. The first was called Traf-o-data. It does not seem to have done as well as Micro-soft.

In Robert's defense, he was skeptical about Artix. I dragged him into it. [3] But there were moments when he was optimistic. And if we, who were 29 and 30 at the time, could get excited about such a thoroughly boneheaded idea, we should not be surprised that hackers aged 21 or 22 are pitching us ideas with little hope of making money.

The Still Life Effect

Why does this happen? Why do good hackers have bad business ideas?

Let's look at our case. One reason we had such a lame idea was that it was the first thing we thought of. I was in New York trying to be a starving artist at the time (the starving part is actually quite easy), so I was haunting galleries anyway. When I learned about the Web, it seemed natural to mix the two. Make Web sites for galleries—that's the ticket!

If you're going to spend years working on something, you'd think it might be wise to spend at least a couple days considering different ideas, instead of going with the first that comes into your head. You'd think. But people don't. In fact, this is a constant problem when you're painting still lifes. You plonk down a bunch of stuff on a table, and maybe spend five or ten minutes rearranging it to look interesting. But you're so impatient to get started painting that ten minutes of rearranging feels very long. So you start painting. Three days later, having spent twenty hours staring at it, you're kicking yourself for having set up such an awkward and boring composition, but by then it's too late.

Part of the problem is that big projects tend to grow out of small ones. You set up a still life to make a quick sketch when you have a spare hour, and days later you're still working on it. I once spent a month painting three versions of a still life I set up in about four minutes. At each point (a day, a week, a month) I thought I'd already put in so much time that it was too late to change.

So the biggest cause of bad ideas is the still life effect: you come up with a random idea, plunge into it, and then at each point (a day, a week, a month) feel you've put so much time into it that this must be *the* idea.

How do we fix that? I don't think we should discard plunging. Plunging into an idea is a good thing. The solution is at the other end: to realize that having invested time in something doesn't make it good.

This is clearest in the case of names. Viaweb was originally called Webgen, but we discovered someone else had a product called that. We were so attached to our name that we offered him *5% of the company* if he'd let us have it. But he wouldn't, so we had to think of another. [4] The best we could do was Viaweb, which we disliked at first. It was like having a new mother. But within three days we loved it, and Webgen sounded lame and old-fashioned.

If it's hard to change something so simple as a name, imagine how hard it is to garbage-collect an idea. A name only has one point of attachment into your head. An idea for a company gets woven into your thoughts. So you must consciously

discount for that. Plunge in, by all means, but remember later to look at your idea in the harsh light of morning and ask: is this something people will pay for? Is this, of all the things we could make, the thing people will pay most for?

Muck

The second mistake we made with Artix is also very common. Putting galleries on the Web seemed cool.

One of the most valuable things my father taught me is an old Yorkshire saying: where there's muck, there's brass. Meaning that unpleasant work pays. And more to the point here, vice versa. Work people like doesn't pay well, for reasons of supply and demand. The most extreme case is developing programming languages, which doesn't pay at all, because people like it so much they do it for free.

When we started Artix, I was still ambivalent about business. I wanted to keep one foot in the art world. Big, big, mistake. Going into business is like a hang-glider launch: you'd better do it wholeheartedly, or not at all. The purpose of a company, and a startup especially, is to make money. You can't have divided loyalties.

Which is not to say that you have to do the most disgusting sort of work, like spamming, or starting a company whose only purpose is patent litigation. What I mean is, if you're starting a company that will do something cool, the aim had better be to make money and maybe be cool, not to be cool and maybe make money.

It's hard enough to make money that you can't do it by accident. Unless it's your first priority, it's unlikely to happen at all.

Hyenas

When I probe our motives with Artix, I see a third mistake: timidity. If you'd proposed at the time that we go into the e-commerce business, we'd have found the idea terrifying. Surely a field like that would be dominated by fearsome startups with five million dollars of VC money each. Whereas we felt pretty sure that we could hold our own in the slightly less competitive business of generating Web sites for art galleries.

We erred ridiculously far on the side of safety. As it turns out, VC-backed startups are not that fearsome. They're too busy trying to spend all that money to get software written. In 1995, the e-commerce business was very competitive as measured in press releases, but not as measured in software. And really it never was. The big fish like Open Market (rest their souls) were just consulting companies pretending to be product companies [5], and the offerings at our end of the market were a couple hundred lines of Perl scripts. Or could have been implemented as a couple hundred lines of Perl; in fact they were probably tens of thousands of lines of C++ or Java. Once we actually took the plunge into e-commerce, it turned out to be surprisingly easy to compete.

So why were we afraid? We felt we were good at programming, but we lacked confidence in our ability to do a mysterious, undifferentiated thing we called "business." In fact there is no such thing as "business." There's selling, promotion, figuring out what people want, deciding how much to charge, customer support,

paying your bills, getting customers to pay you, getting incorporated, raising money, and so on. And the combination is not as hard as it seems, because some tasks (like raising money and getting incorporated) are an O(1) pain in the ass, whether you're big or small, and others (like selling and promotion) depend more on energy and imagination than any kind of special training.

Artix was like a hyena, content to survive on carrion because we were afraid of the lions. Except the lions turned out not to have any teeth, and the business of putting galleries online barely qualified as carrion.

A Familiar Problem

Sum up all these sources of error, and it's no wonder we had such a bad idea for a company. We did the first thing we thought of; we were ambivalent about being in business at all; and we deliberately chose an impoverished market to avoid competition.

Looking at the applications for the Summer Founders Program, I see signs of all three. But the first is by far the biggest problem. Most of the groups applying have not stopped to ask: of all the things we could do, is *this* the one with the best chance of making money?

If they'd already been through their Artix phase, they'd have learned to ask that. After the reception we got from art dealers, we were ready to. This time, we thought, let's make something people want.

Reading the *Wall Street Journal* for a week should give anyone ideas for two or three new startups. The articles are full of descriptions of problems that need to be solved. But most of the applicants don't seem to have looked far for ideas.

We expected the most common proposal to be for multiplayer games. We were not far off: this was the second most common. The most common was some combination of a blog, a calendar, a dating site, and Friendster. Maybe there is some new killer app to be discovered here, but it seems perverse to go poking around in this fog when there are valuable, unsolved problems lying about in the open for anyone to see. Why did no one propose a new scheme for micropayments? An ambitious project, perhaps, but I can't believe we've considered every alternative. And newspapers and magazines are (literally) dying for a solution.

Why did so few applicants really think about what customers want? I think the problem with many, as with people in their early twenties generally, is that they've been trained their whole lives to jump through predefined hoops. They've spent 15-20 years solving problems other people have set for them. And how much time deciding what problems would be good to solve? Two or three course projects? They're good at solving problems, but bad at choosing them.

But that, I'm convinced, is just the effect of training. Or more precisely, the effect of grading. To make grading efficient, everyone has to solve the same problem, and that means it has to be decided in advance. It would be great if schools taught students how to choose problems as well as how to solve them, but I don't know how you'd run such a class in practice.

Copper and Tin

The good news is, choosing problems is something that can be learned. I know that from experience. Hackers can learn to make things customers want. [6]

This is a controversial view. One expert on "entrepreneurship" told me that any startup had to include business people, because only they could focus on what customers wanted. I'll probably alienate this guy forever by quoting him, but I have to risk it, because his email was such a perfect example of this view:

> 80% of MIT spinoffs succeed *provided* they have at least one management person in the team at the start. The business person represents the "voice of the customer" and that's what keeps the engineers and product development on track.

This is, in my opinion, a crock. Hackers are perfectly capable of hearing the voice of the customer without a business person to amplify the signal for them. Larry Page and Sergey Brin were grad students in computer science, which presumably makes them "engineers." Do you suppose Google is only good because they had some business guy whispering in their ears what customers wanted? It seems to me the business guys who did the most for Google were the ones who obligingly flew Altavista into a hillside just as Google was getting started.

The hard part about figuring out what customers want is figuring out that you need to figure it out. But that's something you can learn quickly. It's like seeing the other interpretation of an ambiguous picture. As soon as someone tells you there's a rabbit as well as a duck, it's hard not to see it.

And compared to the sort of problems hackers are used to solving, giving customers what they want is easy. Anyone who can write an optimizing compiler can design a UI that doesn't confuse users, once they *choose* to focus on that problem. And once you apply that kind of brain power to petty but profitable questions, you can create wealth very rapidly.

That's the essence of a startup: having brilliant people do work that's beneath them. Big companies try to hire the right person for the job. Startups win because they don't—because they take people so smart that they would in a big company be doing "research," and set them to work instead on problems of the most immediate and mundane sort. Think Einstein designing refrigerators. [7]

If you want to learn what people want, read Dale Carnegie's *How to Win Friends and Influence People.* [8] When a friend recommended this book, I couldn't believe he was serious. But he insisted it was good, so I read it, and he was right. It deals with the most difficult problem in human experience: how to see things from other people's point of view, instead of thinking only of yourself.

Most smart people don't do that very well. But adding this ability to raw brainpower is like adding tin to copper. The result is bronze, which is so much harder that it seems a different metal.

A hacker who has learned what to make, and not just how to make, is

extraordinarily powerful. And not just at making money: look what a small group of volunteers has achieved with Firefox.

Doing an Artix teaches you to make something people want in the same way that not drinking anything would teach you how much you depend on water. But it would be more convenient for all involved if the Summer Founders didn't learn this on our dime—if they could skip the Artix phase and go right on to make something customers wanted. That, I think, is going to be the real experiment this summer. How long will it take them to grasp this?

We decided we ought to have T-Shirts for the SFP, and we'd been thinking about what to print on the back. Till now we'd been planning to use

>   If you can read this, I should be working.

but now we've decided it's going to be

>   Make something people want.

Notes

[1] SFP applicants: please don't assume that not being accepted means we think your idea is bad. Because we want to keep the number of startups small this first summer, we're going to have to turn down some good proposals too.

[2] Dealers try to give each customer the impression that the stuff they're showing him is something special that only a few people have seen, when in fact it may have been sitting in their racks for years while they tried to unload it on buyer after buyer.

[3] On the other hand, he was skeptical about Viaweb too. I have a precise measure of that, because at one point in the first couple months we made a bet: if he ever made a million dollars out of Viaweb, he'd get his ear pierced. We didn't let him off, either.

[4] I wrote a program to generate all the combinations of "Web" plus a three letter word. I learned from this that most three letter words are bad: Webpig, Webdog, Webfat, Webzit, Webfug. But one of them was Webvia; I swapped them to make Viaweb.

[5] It's much easier to sell services than a product, just as it's easier to make a living playing at weddings than by selling recordings. But the margins are greater on products. So during the Bubble a lot of companies used consulting to generate revenues they could attribute to the sale of products, because it made a better story for an IPO.

[6] Trevor Blackwell presents the following recipe for a startup: "Watch people who have money to spend, see what they're wasting their time on, cook up a solution, and try selling it to them. It's surprising how small a problem can be and still provide a profitable market for a solution."

[7] You need to offer especially large rewards to get great people to do tedious work. That's why startups always pay equity rather than just salary.

[8] Buy an old copy from the 1940s or 50s instead of the current edition, which has been rewritten to suit present fashions. The original edition contained a few unPC ideas, but it's always better to read an original book, bearing in mind that it's a book from a past era, than to read a new version sanitized for your protection.

Thanks to Bill Birch, Trevor Blackwell, Jessica Livingston, and Robert Morris for reading drafts of this.

■

Russian Translation

■

Italian Translation

■

Japanese Translation

If you liked this, you may also like ***Hackers & Painters***.

The Submarine

April 2005

"Suits make a corporate comeback," says the *New York Times*. Why does this sound familiar? Maybe because the suit was also back in February, September 2004, June 2004, March 2004, September 2003, November 2002, April 2002, and February 2002.

Why do the media keep running stories saying suits are back? Because PR firms tell them to. One of the most surprising things I discovered during my brief business career was the existence of the PR industry, lurking like a huge, quiet submarine beneath the news. Of the stories you read in traditional media that aren't about politics, crimes, or disasters, more than half probably come from PR firms.

I know because I spent years hunting such "press hits." Our startup spent its entire marketing budget on PR: at a time when we were assembling our own computers to save money, we were paying a PR firm $16,000 a month. And they were worth it. PR is the news equivalent of search engine optimization; instead of buying ads, which readers ignore, you get yourself inserted directly into the stories. [1]

Our PR firm was one of the best in the business. In 18 months, they got press hits in over 60 different publications. And we weren't the only ones they did great things for. In 1997 I got a call from another startup founder considering hiring them to promote his company. I told him they were PR gods, worth every penny of their outrageous fees. But I remember thinking his company's name was odd. Why call an auction site "eBay"?

Symbiosis

PR is not dishonest. Not quite. In fact, the reason the best PR firms are so effective is precisely that they aren't dishonest. They give reporters genuinely valuable information. A good PR firm won't bug reporters just because the client tells them to; they've worked hard to build their credibility with reporters, and they don't want to destroy it by feeding them mere propaganda.

If anyone is dishonest, it's the reporters. The main reason PR firms exist is that reporters are lazy. Or, to put it more nicely, overworked. Really they ought to be

out there digging up stories for themselves. But it's so tempting to sit in their offices and let PR firms bring the stories to them. After all, they know good PR firms won't lie to them.

A good flatterer doesn't lie, but tells his victim selective truths (what a nice color your eyes are). Good PR firms use the same strategy: they give reporters stories that are true, but whose truth favors their clients.

For example, our PR firm often pitched stories about how the Web let small merchants compete with big ones. This was perfectly true. But the reason reporters ended up writing stories about this particular truth, rather than some other one, was that small merchants were our target market, and we were paying the piper.

Different publications vary greatly in their reliance on PR firms. At the bottom of the heap are the trade press, who make most of their money from advertising and would give the magazines away for free if advertisers would let them. [2] The average trade publication is a bunch of ads, glued together by just enough articles to make it look like a magazine. They're so desperate for "content" that some will print your press releases almost verbatim, if you take the trouble to write them to read like articles.

At the other extreme are publications like the *New York Times* and the *Wall Street Journal*. Their reporters do go out and find their own stories, at least some of the time. They'll listen to PR firms, but briefly and skeptically. We managed to get press hits in almost every publication we wanted, but we never managed to crack the print edition of the *Times*. [3]

The weak point of the top reporters is not laziness, but vanity. You don't pitch stories to them. You have to approach them as if you were a specimen under their all-seeing microscope, and make it seem as if the story you want them to run is something they thought of themselves.

Our greatest PR coup was a two-part one. We estimated, based on some fairly informal math, that there were about 5000 stores on the Web. We got one paper to print this number, which seemed neutral enough. But once this "fact" was out there in print, we could quote it to other publications, and claim that with 1000 users we had 20% of the online store market.

This was roughly true. We really did have the biggest share of the online store market, and 5000 was our best guess at its size. But the way the story appeared in the press sounded a lot more definite.

Reporters like definitive statements. For example, many of the stories about Jeremy Jaynes's conviction say that he was one of the 10 worst spammers. This "fact" originated in Spamhaus's ROKSO list, which I think even Spamhaus would admit is a rough guess at the top spammers. The first stories about Jaynes cited this source, but now it's simply repeated as if it were part of the indictment. [4]

All you can say with certainty about Jaynes is that he was a fairly big spammer. But reporters don't want to print vague stuff like "fairly big." They want statements with punch, like "top ten." And PR firms give them what they want. Wearing suits, we're told, will make us 3.6 percent more productive.

Buzz

Where the work of PR firms really does get deliberately misleading is in the generation of "buzz." They usually feed the same story to several different publications at once. And when readers see similar stories in multiple places, they think there is some important trend afoot. Which is exactly what they're supposed to think.

When Windows 95 was launched, people waited outside stores at midnight to buy the first copies. None of them would have been there without PR firms, who generated such a buzz in the news media that it became self-reinforcing, like a nuclear chain reaction.

I doubt PR firms realize it yet, but the Web makes it possible to track them at work. If you search for the obvious phrases, you turn up several efforts over the years to place stories about the return of the suit. For example, the Reuters article that got picked up by USA Today in September 2004. "The suit is back," it begins.

Trend articles like this are almost always the work of PR firms. Once you know how to read them, it's straightforward to figure out who the client is. With trend stories, PR firms usually line up one or more "experts" to talk about the industry generally. In this case we get three: the NPD Group, the creative director of GQ, and a research director at Smith Barney. [5] When you get to the end of the experts, look for the client. And bingo, there it is: The Men's Wearhouse.

Not surprising, considering The Men's Wearhouse was at that moment running ads saying "The Suit is Back." Talk about a successful press hit-- a wire service article whose first sentence is your own ad copy.

The secret to finding other press hits from a given pitch is to realize that they all started from the same document back at the PR firm. Search for a few key phrases and the names of the clients and the experts, and you'll turn up other variants of this story.

Casual fridays are out and dress codes are in writes Diane E. Lewis in *The Boston Globe*. In a remarkable coincidence, Ms. Lewis's industry contacts also include the creative director of GQ.

Ripped jeans and T-shirts are out, writes Mary Kathleen Flynn in *US News & World Report*. And *she too* knows the creative director of GQ.

Men's suits are back writes Nicole Ford in Sexbuzz.Com ("the ultimate men's

entertainment magazine").

[Dressing down loses appeal as men suit up at the office](#) writes Tenisha Mercer of *The Detroit News*.

Now that so many news articles are online, I suspect you could find a similar pattern for most trend stories placed by PR firms. I propose we call this new sport "PR diving," and I'm sure there are far more striking examples out there than this clump of five stories.

Online

After spending years chasing them, it's now second nature to me to recognize press hits for what they are. But before we hired a PR firm I had no idea where articles in the mainstream media came from. I could tell a lot of them were crap, but I didn't realize why.

Remember the exercises in critical reading you did in school, where you had to look at a piece of writing and step back and ask whether the author was telling the whole truth? If you really want to be a critical reader, it turns out you have to step back one step further, and ask not just whether the author is telling the truth, but *why he's writing about this subject at all.*

Online, the answer tends to be a lot simpler. Most people who publish online write what they write for the simple reason that they want to. You can't see the fingerprints of PR firms all over the articles, as you can in so many print publications-- which is one of the reasons, though they may not consciously realize it, that readers trust bloggers more than *Business Week*.

I was talking recently to a friend who works for a big newspaper. He thought the print media were in serious trouble, and that they were still mostly in denial about it. "They think the decline is cyclic," he said. "Actually it's structural."

In other words, the readers are leaving, and they're not coming back.

Why? I think the main reason is that the writing online is more honest. Imagine how incongruous the *New York Times* article about suits would sound if you read it in a blog:

> The urge to look corporate-- sleek, commanding, prudent, yet with just a touch of hubris on your well-cut sleeve-- is an unexpected development in a time of business disgrace.

The problem with this article is not just that it originated in a PR firm. The whole tone is bogus. This is the tone of someone writing down to their audience.

Whatever its flaws, the writing you find online is authentic. It's not mystery meat cooked up out of scraps of pitch letters and press releases, and pressed into molds

of zippy journalese. It's people writing what they think.

I didn't realize, till there was an alternative, just how artificial most of the writing in the mainstream media was. I'm not saying I used to believe what I read in *Time* and *Newsweek*. Since high school, at least, I've thought of magazines like that more as guides to what ordinary people were being [told](#) to think than as sources of information. But I didn't realize till the last few years that writing for publication didn't have to mean writing that way. I didn't realize you could write as candidly and informally as you would if you were writing to a friend.

Readers aren't the only ones who've noticed the change. The PR industry has too. A hilarious [article](#) on the site of the PR Society of America gets to the heart of the matter:

> Bloggers are sensitive about becoming mouthpieces for other organizations and companies, which is the reason they began blogging in the first place.

PR people fear bloggers for the same reason readers like them. And that means there may be a struggle ahead. As this new kind of writing draws readers away from traditional media, we should be prepared for whatever PR mutates into to compensate. When I think how hard PR firms work to score press hits in the traditional media, I can't imagine they'll work any less hard to feed stories to bloggers, if they can figure out how.

Notes

[1] PR has at least one beneficial feature: it favors small companies. If PR didn't work, the only alternative would be to advertise, and only big companies can afford that.

[2] Advertisers pay less for ads in free publications, because they assume readers ignore something they get for free. This is why so many trade publications nominally have a cover price and yet give away free subscriptions with such abandon.

[3] Different sections of the *Times* vary so much in their standards that they're practically different papers. Whoever fed the style section reporter this story about suits coming back would have been sent packing by the regular news reporters.

[4] The most striking example I know of this type is the "fact" that the Internet worm of 1988 infected 6000 computers. I was there when it was cooked up, and this was the recipe: someone guessed that there were about 60,000 computers attached to the Internet, and that the worm might have infected ten percent of them.

Actually no one knows how many computers the worm infected, because the remedy was to reboot them, and this destroyed all traces. But people like numbers. And so this one is now replicated all over the Internet, like a little worm of its own.

[5] Not all were necessarily supplied by the PR firm. Reporters sometimes call a few additional sources on their own, like someone adding a few fresh vegetables to a can of soup.

Thanks to Ingrid Basset, Trevor Blackwell, Sarah Harlin, Jessica Livingston, Jackie McDonough, Robert Morris, and Aaron Swartz (who also found the PRSA article) for reading drafts of this.

Correction: Earlier versions used a recent *Business Week* article mentioning del.icio.us as an example of a press hit, but Joshua Schachter tells me it was spontaneous.

The Web is a Writing Environment

A Sell-Out's Tale

How to Pitch Bloggers

Blogging for Milk

7 Habits of Highly Effective Blog PR

PR People Need To Learn To Deal With New Gatekeepers

- Marqui Blogosphere Program

- PR Watch

- Real Men Exfoliate

- How the News is Made

- January 2006: The suit is back yet again

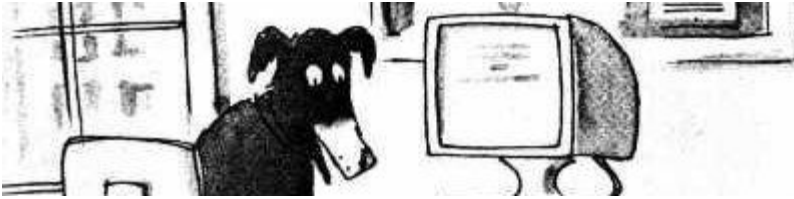- The Decline of the Tie

- Japanese Translation

If you liked this, you may also like ***Hackers & Painters***.

Hiring is Obsolete

May 2005

(This essay is derived from a talk at the Berkeley CSUA.)

The three big powers on the Internet now are Yahoo, Google, and Microsoft. Average age of their founders: 24. So it is pretty well established now that grad students can start successful companies. And if grad students can do it, why not undergrads?

Like everything else in technology, the cost of starting a startup has decreased dramatically. Now it's so low that it has disappeared into the noise. The main cost of starting a Web-based startup is food and rent. Which means it doesn't cost much more to start a company than to be a total slacker. You can probably start a startup on ten thousand dollars of seed funding, if you're prepared to live on ramen.

The less it costs to start a company, the less you need the permission of investors to do it. So a lot of people will be able to start companies now who never could have before.

The most interesting subset may be those in their early twenties. I'm not so excited about founders who have everything investors want except intelligence, or everything except energy. The most promising group to be liberated by the new, lower threshold are those who have everything investors want except experience.

Market Rate

I once claimed that nerds were unpopular in secondary school mainly because they had better things to do than work full-time at being popular. Some said I was just telling people what they wanted to hear. Well, I'm now about to do that in a spectacular way: I think undergraduates are undervalued.

Or more precisely, I think few realize the huge spread in the value of 20 year olds. Some, it's true, are not very capable. But others are more capable than all but a handful of 30 year olds. [1]

Till now the problem has always been that it's difficult to pick them out. Every VC in the world, if they could go back in time, would try to invest in Microsoft. But which would have then? How many would have understood that this particular 19 year old was Bill Gates?

It's hard to judge the young because (a) they change rapidly, (b) there is great variation between them, and (c) they're individually inconsistent. That last one is a big problem. When you're young, you occasionally say and do stupid things even when you're smart. So if the algorithm is to filter out people who say stupid things, as many investors and employers unconsciously do, you're going to get a lot of false positives.

Most organizations who hire people right out of college are only aware of the average value of 22 year olds, which is not that high. And so the idea for most of the twentieth century was that everyone had to begin as a trainee in some [entry-level](#) job. Organizations realized there was a lot of variation in the incoming stream, but instead of pursuing this thought they tended to suppress it, in the belief that it was good for even the most promising kids to start at the bottom, so they didn't get swelled heads.

The most productive young people will *always* be undervalued by large organizations, because the young have no performance to measure yet, and any error in guessing their ability will tend toward the mean.

What's an especially productive 22 year old to do? One thing you can do is go over the heads of organizations, directly to the users. Any company that hires you is, economically, acting as a proxy for the customer. The rate at which they value you (though they may not consciously realize it) is an attempt to guess your value to the user. But there's a way to appeal their judgement. If you want, you can opt to be valued directly by users, by starting your own company.

The market is a lot more discerning than any employer. And it is completely non-discriminatory. On the Internet, nobody knows you're a dog. And more to the point, nobody knows you're 22. All users care about is whether your site or software gives them what they want. They don't care if the person behind it is a high school kid.

If you're really productive, why not make employers pay market rate for you? Why go work as an ordinary employee for a big company, when you could start a startup and make them buy it to get you?

When most people hear the word "startup," they think of the famous ones that have gone public. But most startups that succeed do it by getting bought. And usually the acquirer doesn't just want the technology, but the people who created it as well.

Often big companies buy startups before they're profitable. Obviously in such cases they're not after revenues. What they want is the development team and the software they've built so far. When a startup gets bought for 2 or 3 million six months in, it's really more of a hiring bonus than an acquisition.

I think this sort of thing will happen more and more, and that it will be better for everyone. It's obviously better for the people who start the startup, because they

get a big chunk of money up front. But I think it will be better for the acquirers too. The central problem in big companies, and the main reason they're so much less productive than small companies, is the difficulty of valuing each person's work. Buying larval startups solves that problem for them: the acquirer doesn't pay till the developers have proven themselves. Acquirers are protected on the downside, but still get most of the upside.

Product Development

Buying startups also solves another problem afflicting big companies: they can't do product development. Big companies are good at extracting the value from existing products, but bad at creating new ones.

Why? It's worth studying this phenomenon in detail, because this is the raison d'etre of startups.

To start with, most big companies have some kind of turf to protect, and this tends to warp their development decisions. For example, Web-based applications are hot now, but within Microsoft there must be a lot of ambivalence about them, because the very idea of Web-based software threatens the desktop. So any Web-based application that Microsoft ends up with, will probably, like Hotmail, be something developed outside the company.

Another reason big companies are bad at developing new products is that the kind of people who do that tend not to have much power in big companies (unless they happen to be the CEO). Disruptive technologies are developed by disruptive people. And they either don't work for the big company, or have been outmaneuvered by yes-men and have comparatively little influence.

Big companies also lose because they usually only build one of each thing. When you only have one Web browser, you can't do anything really risky with it. If ten different startups design ten different Web browsers and you take the best, you'll probably get something better.

The more general version of this problem is that there are too many new ideas for companies to explore them all. There might be 500 startups right now who think they're making something Microsoft might buy. Even Microsoft probably couldn't manage 500 development projects in-house.

Big companies also don't pay people the right way. People developing a new product at a big company get paid roughly the same whether it succeeds or fails. People at a startup expect to get rich if the product succeeds, and get nothing if it fails. [2] So naturally the people at the startup work a lot harder.

The mere bigness of big companies is an obstacle. In startups, developers are often forced to talk directly to users, whether they want to or not, because there is no one else to do sales and support. It's painful doing sales, but you learn much more from trying to sell people something than reading what they said in focus groups.

And then of course, big companies are bad at product development because they're bad at everything. Everything happens slower in big companies than small ones, and product development is something that has to happen fast, because you have to go through a lot of iterations to get something good.

Trend

I think the trend of big companies buying startups will only accelerate. One of the biggest remaining obstacles is pride. Most companies, at least unconsciously, feel they ought to be able to develop stuff in house, and that buying startups is to some degree an admission of failure. And so, as people generally do with admissions of failure, they put it off for as long as possible. That makes the acquisition very expensive when it finally happens.

What companies should do is go out and discover startups when they're young, before VCs have puffed them up into something that costs hundreds of millions to acquire. Much of what VCs add, the acquirer doesn't need anyway.

Why don't acquirers try to predict the companies they're going to have to buy for hundreds of millions, and grab them early for a tenth or a twentieth of that? Because they can't predict the winners in advance? If they're only paying a twentieth as much, they only have to predict a twentieth as well. Surely they can manage that.

I think companies that acquire technology will gradually learn to go after earlier stage startups. They won't necessarily buy them outright. The solution may be some hybrid of investment and acquisition: for example, to buy a chunk of the company and get an option to buy the rest later.

When companies buy startups, they're effectively fusing recruiting and product development. And I think that's more efficient than doing the two separately, because you always get people who are really committed to what they're working on.

Plus this method yields teams of developers who already work well together. Any conflicts between them have been ironed out under the very hot iron of running a startup. By the time the acquirer gets them, they're finishing one another's sentences. That's valuable in software, because so many bugs occur at the boundaries between different people's code.

Investors

The increasing cheapness of starting a company doesn't just give hackers more power relative to employers. It also gives them more power relative to investors.

The conventional wisdom among VCs is that hackers shouldn't be allowed to run their own companies. The founders are supposed to accept MBAs as their bosses, and themselves take on some title like Chief Technical Officer. There may be cases where this is a good idea. But I think founders will increasingly be able to push back in the matter of control, because they just don't need the investors' money as much as they used to.

Startups are a comparatively new phenomenon. Fairchild Semiconductor is considered the first VC-backed startup, and they were founded in 1959, less than fifty years ago. Measured on the time scale of social change, what we have now is pre-beta. So we shouldn't assume the way startups work now is the way they have to work.

Fairchild needed a lot of money to get started. They had to build actual factories. What does the first round of venture funding for a Web-based startup get spent on today? More money can't get software written faster; it isn't needed for facilities, because those can now be quite cheap; all money can really buy you is sales and marketing. A sales force is worth something, I'll admit. But marketing is increasingly irrelevant. On the Internet, anything genuinely good will spread by word of mouth.

Investors' power comes from money. When startups need less money, investors have less power over them. So future founders may not have to accept new CEOs if they don't want them. The VCs will have to be dragged kicking and screaming down this road, but like many things people have to be dragged kicking and screaming toward, it may actually be good for them.

Google is a sign of the way things are going. As a condition of funding, their investors insisted they hire someone old and experienced as CEO. But from what I've heard the founders didn't just give in and take whoever the VCs wanted. They delayed for an entire year, and when they did finally take a CEO, they chose a guy with a PhD in computer science.

It sounds to me as if the founders are still the most powerful people in the company, and judging by Google's performance, their youth and inexperience doesn't seem to have hurt them. Indeed, I suspect Google has done better than they would have if the founders had given the VCs what they wanted, when they wanted it, and let some MBA take over as soon as they got their first round of funding.

I'm not claiming the business guys installed by VCs have no value. Certainly they have. But they don't need to become the founders' bosses, which is what that title CEO means. I predict that in the future the executives installed by VCs will increasingly be COOs rather than CEOs. The founders will run engineering directly, and the rest of the company through the COO.

The Open Cage

With both employers and investors, the balance of power is slowly shifting towards the young. And yet they seem the last to realize it. Only the most ambitious undergrads even consider starting their own company when they graduate. Most just want to get a job.

Maybe this is as it should be. Maybe if the idea of starting a startup is intimidating, you filter out the uncommitted. But I suspect the filter is set a little too high. I think there are people who could, if they tried, start successful startups, and who instead let themselves be swept into the intake ducts of big companies.

Have you ever noticed that when animals are let out of cages, they don't always realize at first that the door's open? Often they have to be poked with a stick to get them out. Something similar happened with blogs. People could have been publishing online in 1995, and yet blogging has only really taken off in the last couple years. In 1995 we thought only professional writers were entitled to publish their ideas, and that anyone else who did was a crank. Now publishing online is becoming so popular that everyone wants to do it, even print journalists. But blogging has not taken off recently because of any technical innovation; it just took eight years for everyone to realize the cage was open.

I think most undergrads don't realize yet that the economic cage is open. A lot have been told by their parents that the route to success is to get a good job. This was true when their parents were in college, but it's less true now. The route to success is to build something valuable, and you don't have to be working for an existing company to do that. Indeed, you can often do it better if you're not.

When I talk to undergrads, what surprises me most about them is how conservative they are. Not politically, of course. I mean they don't seem to want to take risks. This is a mistake, because the younger you are, the more risk you can take.

Risk

Risk and reward are always proportionate. For example, stocks are riskier than bonds, and over time always have greater returns. So why does anyone invest in bonds? The catch is that phrase "over time." Stocks will generate greater returns over thirty years, but they might lose value from year to year. So what you should invest in depends on how soon you need the money. If you're young, you should take the riskiest investments you can find.

All this talk about investing may seem very theoretical. Most undergrads probably have more debts than assets. They may feel they have nothing to invest. But that's not true: they have their time to invest, and the same rule about risk applies there. Your early twenties are exactly the time to take insane career risks.

The reason risk is always proportionate to reward is that market forces make it so. People will pay extra for stability. So if you choose stability-- by buying bonds, or by going to work for a big company-- it's going to cost you.

Riskier career moves pay better on average, because there is less demand for them. Extreme choices like starting a startup are so frightening that most people won't even try. So you don't end up having as much competition as you might expect, considering the prizes at stake.

The math is brutal. While perhaps 9 out of 10 startups fail, the one that succeeds will pay the founders more than 10 times what they would have made in an ordinary job. [3] That's the sense in which startups pay better "on average."

Remember that. If you start a startup, you'll probably fail. Most startups fail. It's the nature of the business. But it's not necessarily a mistake to try something that has a 90% chance of failing, if you can afford the risk. Failing at 40, when you have a family to support, could be serious. But if you fail at 22, so what? If you try to start a startup right out of college and it tanks, you'll end up at 23 broke and a lot smarter. Which, if you think about it, is roughly what you hope to get from a graduate program.

Even if your startup does tank, you won't harm your prospects with employers. To make sure I asked some friends who work for big companies. I asked managers at Yahoo, Google, Amazon, Cisco and Microsoft how they'd feel about two candidates, both 24, with equal ability, one who'd tried to start a startup that tanked, and another who'd spent the two years since college working as a developer at a big company. Every one responded that they'd prefer the guy who'd tried to start his own company. Zod Nazem, who's in charge of engineering at Yahoo, said:

> I actually put more value on the guy with the failed startup. And you can quote me!

So there you have it. Want to get hired by Yahoo? Start your own company.

The Man is the Customer

If even big employers think highly of young hackers who start companies, why don't more do it? Why are undergrads so conservative? I think it's because they've spent so much time in institutions.

The first twenty years of everyone's life consists of being piped from one institution to another. You probably didn't have much choice about the secondary schools you went to. And after high school it was probably understood that you were supposed to go to college. You may have had a few different colleges to choose between, but they were probably pretty similar. So by this point you've been riding on a subway line for twenty years, and the next stop seems to be a job.

Actually college is where the line ends. Superficially, going to work for a company may feel like just the next in a series of institutions, but underneath, everything is different. The end of school is the fulcrum of your life, the point where you go from net consumer to net producer.

The other big change is that now, you're steering. You can go anywhere you want. So it may be worth standing back and understanding what's going on, instead of just doing the default thing.

All through college, and probably long before that, most undergrads have been thinking about what employers want. But what really matters is what customers want, because they're the ones who give employers the money to pay you.

So instead of thinking about what employers want, you're probably better off thinking directly about what users want. To the extent there's any difference between the two, you can even use that to your advantage if you start a company of your own. For example, big companies like docile conformists. But this is merely an artifact of their bigness, not something customers need.

Grad School

I didn't consciously realize all this when I was graduating from college-- partly because I went straight to grad school. Grad school can be a pretty good deal, even if you think of one day starting a startup. You can start one when you're done, or even pull the ripcord part way through, like the founders of Yahoo and Google.

Grad school makes a good launch pad for startups, because you're collected together with a lot of smart people, and you have bigger chunks of time to work on your own projects than an undergrad or corporate employee would. As long as you have a fairly tolerant advisor, you can take your time developing an idea before

turning it into a company. David Filo and Jerry Yang started the Yahoo directory in February 1994 and were getting a million hits a day by the fall, but they didn't actually drop out of grad school and start a company till March 1995.

You could also try the startup first, and if it doesn't work, then go to grad school. When startups tank they usually do it fairly quickly. Within a year you'll know if you're wasting your time.

If it fails, that is. If it succeeds, you may have to delay grad school a little longer. But you'll have a much more enjoyable life once there than you would on a regular grad student stipend.

Experience

Another reason people in their early twenties don't start startups is that they feel they don't have enough experience. Most investors feel the same.

I remember hearing a lot of that word "experience" when I was in college. What do people really mean by it? Obviously it's not the experience itself that's valuable, but something it changes in your brain. What's different about your brain after you have "experience," and can you make that change happen faster?

I now have some data on this, and I can tell you what tends to be missing when people lack experience. I've said that every startup needs three things: to start with good people, to make something users want, and not to spend too much money. It's the middle one you get wrong when you're inexperienced. There are plenty of undergrads with enough technical skill to write good software, and undergrads are not especially prone to waste money. If they get something wrong, it's usually not realizing they have to make something people want.

This is not exclusively a failing of the young. It's common for startup founders of all ages to build things no one wants.

Fortunately, this flaw should be easy to fix. If undergrads were all bad programmers, the problem would be a lot harder. It can take years to learn how to program. But I don't think it takes years to learn how to make things people want. My hypothesis is that all you have to do is smack hackers on the side of the head and tell them: Wake up. Don't sit here making up a priori theories about what users need. Go find some users and see what they need.

Most successful startups not only do something very specific, but solve a problem people already know they have.

The big change that "experience" causes in your brain is learning that you need to solve people's problems. Once you grasp that, you advance quickly to the next step, which is figuring out what those problems are. And that takes some effort, because the way software actually gets used, especially by the people who pay the most for it, is not at all what you might expect. For example, the stated purpose of

Powerpoint is to present ideas. Its real role is to overcome people's fear of public speaking. It allows you to give an impressive-looking talk about nothing, and it causes the audience to sit in a dark room looking at slides, instead of a bright one looking at you.

This kind of thing is out there for anyone to see. The key is to know to look for it-- to realize that having an idea for a startup is not like having an idea for a class project. The goal in a startup is not to write a cool piece of software. It's to make something people want. And to do that you have to look at users-- forget about hacking, and just look at users. This can be quite a mental adjustment, because little if any of the software you write in school even has users.

A few steps before a Rubik's Cube is solved, it still looks like a mess. I think there are a lot of undergrads whose brains are in a similar position: they're only a few steps away from being able to start successful startups, if they wanted to, but they don't realize it. They have more than enough technical skill. They just haven't realized yet that the way to create wealth is to make what users want, and that employers are just proxies for users in which risk is pooled.

If you're young and smart, you don't need either of those. You don't need someone else to tell you what users want, because you can figure it out yourself. And you don't want to pool risk, because the younger you are, the more risk you should take.

A Public Service Message

I'd like to conclude with a joint message from me and your parents. Don't drop out of college to start a startup. There's no rush. There will be plenty of time to start companies after you graduate. In fact, it may be just as well to go work for an existing company for a couple years after you graduate, to learn how companies work.

And yet, when I think about it, I can't imagine telling Bill Gates at 19 that he should wait till he graduated to start a company. He'd have told me to get lost. And could I have honestly claimed that he was harming his future-- that he was learning less by working at ground zero of the microcomputer revolution than he would have if he'd been taking classes back at Harvard? No, probably not.

And yes, while it is probably true that you'll learn some valuable things by going to work for an existing company for a couple years before starting your own, you'd learn a thing or two running your own company during that time too.

The advice about going to work for someone else would get an even colder reception from the 19 year old Bill Gates. So I'm supposed to finish college, then go work for another company for two years, and then I can start my own? I have to wait till I'm 23? That's *four years*. That's more than twenty percent of my life so far. Plus in four years it will be way too late to make money writing a Basic interpreter for the Altair.

And he'd be right. The Apple II was launched just two years later. In fact, if Bill had finished college and gone to work for another company as we're suggesting, he might well have gone to work for Apple. And while that would probably have been better for all of us, it wouldn't have been better for him.

So while I stand by our responsible advice to finish college and then go work for a while before starting a startup, I have to admit it's one of those things the old tell the young, but don't expect them to listen to. We say this sort of thing mainly so we can claim we warned you. So don't say I didn't warn you.

Notes

[1] The average B-17 pilot in World War II was in his early twenties. (Thanks to Tad Marko for pointing this out.)

[2] If a company tried to pay employees this way, they'd be called unfair. And yet when they buy some startups and not others, no one thinks of calling that unfair.

[3] The 1/10 success rate for startups is a bit of an urban legend. It's suspiciously neat. My guess is the odds are slightly worse.

Thanks to Jessica Livingston for reading drafts of this, to the friends I promised anonymity to for their opinions about hiring, and to Karen Nguyen and the Berkeley CSUA for organizing this talk.

■

[Russian Translation](#)

■

[Romanian Translation](#)

■

[Japanese Translation](#)

If you liked this, you may also like ***[Hackers & Painters](#)***.

What Business Can Learn from Open Source

August 2005

(This essay is derived from a talk at Oscon 2005.)

Lately companies have been paying more attention to open source. Ten years ago there seemed a real danger Microsoft would extend its monopoly to servers. It seems safe to say now that open source has prevented that. A recent survey found 52% of companies are replacing Windows servers with Linux servers. [1]

More significant, I think, is *which* 52% they are. At this point, anyone proposing to run Windows on servers should be prepared to explain what they know about servers that Google, Yahoo, and Amazon don't.

But the biggest thing business has to learn from open source is not about Linux or Firefox, but about the forces that produced them. Ultimately these will affect a lot more than what software you use.

We may be able to get a fix on these underlying forces by triangulating from open source and blogging. As you've probably noticed, they have a lot in common.

Like open source, blogging is something people do themselves, for free, because they enjoy it. Like open source hackers, bloggers compete with people working for money, and often win. The method of ensuring quality is also the same: Darwinian. Companies ensure quality through rules to prevent employees from screwing up. But you don't need that when the audience can communicate with one another. People just produce whatever they want; the good stuff spreads, and the bad gets ignored. And in both cases, feedback from the audience improves the best work.

Another thing blogging and open source have in common is the Web. People have always been willing to do great work for free, but before the Web it was harder to reach an audience or collaborate on projects.

Amateurs

I think the most important of the new principles business has to learn is that people work a lot harder on stuff they like. Well, that's news to no one. So how can

I claim business has to learn it? When I say business doesn't know this, I mean the structure of business doesn't reflect it.

Business still reflects an older model, exemplified by the French word for working: *travailler*. It has an English cousin, travail, and what it means is torture. [2]

This turns out not to be the last word on work, however. As societies get richer, they learn something about work that's a lot like what they learn about diet. We know now that the healthiest diet is the one our peasant ancestors were forced to eat because they were poor. Like rich food, idleness only seems desirable when you don't get enough of it. I think we were designed to work, just as we were designed to eat a certain amount of fiber, and we feel bad if we don't.

There's a name for people who work for the love of it: amateurs. The word now has such bad connotations that we forget its etymology, though it's staring us in the face. "Amateur" was originally rather a complimentary word. But the thing to be in the twentieth century was professional, which amateurs, by definition, are not.

That's why the business world was so surprised by one lesson from open source: that people working for love often surpass those working for money. Users don't switch from Explorer to Firefox because they want to hack the source. They switch because it's a better browser.

It's not that Microsoft isn't trying. They know controlling the browser is one of the keys to retaining their monopoly. The problem is the same they face in operating systems: they can't pay people enough to build something better than a group of inspired hackers will build for free.

I suspect professionalism was always overrated-- not just in the literal sense of working for money, but also connotations like formality and detachment. Inconceivable as it would have seemed in, say, 1970, I think professionalism was largely a fashion, driven by conditions that happened to exist in the twentieth century.

One of the most powerful of those was the existence of "channels." Revealingly, the same term was used for both products and information: there were distribution channels, and TV and radio channels.

It was the narrowness of such channels that made professionals seem so superior to amateurs. There were only a few jobs as professional journalists, for example, so competition ensured the average journalist was fairly good. Whereas anyone can express opinions about current events in a bar. And so the average person expressing his opinions in a bar sounds like an idiot compared to a journalist writing about the subject.

On the Web, the barrier for publishing your ideas is even lower. You don't have to buy a drink, and they even let kids in. Millions of people are publishing online, and

the average level of what they're writing, as you might expect, is not very good. This has led some in the media to conclude that blogs don't present much of a threat-- that blogs are just a fad.

Actually, the fad is the word "blog," at least the way the print media now use it. What they mean by "blogger" is not someone who publishes in a weblog format, but anyone who publishes online. That's going to become a problem as the Web becomes the default medium for publication. So I'd like to suggest an alternative word for someone who publishes online. How about "writer?"

Those in the print media who dismiss the writing online because of its low average quality are missing an important point: no one reads the *average* blog. In the old world of channels, it meant something to talk about average quality, because that's what you were getting whether you liked it or not. But now you can read any writer you want. So the average quality of writing online isn't what the print media are competing against. They're competing against the best writing online. And, like Microsoft, they're losing.

I know that from my own experience as a reader. Though most print publications are online, I probably read two or three articles on individual people's sites for every one I read on the site of a newspaper or magazine.

And when I read, say, New York Times stories, I never reach them through the Times front page. Most I find through aggregators like Google News or Slashdot or Delicious. Aggregators show how much better you can do than the channel. The New York Times front page is a list of articles written by people who work for the New York Times. Delicious is a list of articles that are interesting. And it's only now that you can see the two side by side that you notice how little overlap there is.

Most articles in the print media are boring. For example, the president notices that a majority of voters now think invading Iraq was a mistake, so he makes an address to the nation to drum up support. Where is the man bites dog in that? I didn't hear the speech, but I could probably tell you exactly what he said. A speech like that is, in the most literal sense, not news: there is nothing *new* in it. [3]

Nor is there anything new, except the names and places, in most "news" about things going wrong. A child is abducted; there's a tornado; a ferry sinks; someone gets bitten by a shark; a small plane crashes. And what do you learn about the world from these stories? Absolutely nothing. They're outlying data points; what makes them gripping also makes them irrelevant.

As in software, when professionals produce such crap, it's not surprising if amateurs can do better. Live by the channel, die by the channel: if you depend on an oligopoly, you sink into bad habits that are hard to overcome when you suddenly get competition. [4]

Workplaces

Another thing blogs and open source software have in common is that they're often made by people working at home. That may not seem surprising. But it should be. It's the architectural equivalent of a home-made aircraft shooting down an F-18. Companies spend millions to build office buildings for a single purpose: to be a place to work. And yet people working in their own homes, which aren't even designed to be workplaces, end up being more productive.

This proves something a lot of us have suspected. The average office is a miserable place to get work done. And a lot of what makes offices bad are the very qualities we associate with professionalism. The sterility of offices is supposed to suggest efficiency. But suggesting efficiency is a different thing from actually being efficient.

The atmosphere of the average workplace is to productivity what flames painted on the side of a car are to speed. And it's not just the way offices look that's bleak. The way people act is just as bad.

Things are different in a startup. Often as not a startup begins in an apartment. Instead of matching beige cubicles they have an assortment of furniture they bought used. They work odd hours, wearing the most casual of clothing. They look at whatever they want online without worrying whether it's "work safe." The cheery, bland language of the office is replaced by wicked humor. And you know what? The company at this stage is probably the most productive it's ever going to be.

Maybe it's not a coincidence. Maybe some aspects of professionalism are actually a net lose.

To me the most demoralizing aspect of the traditional office is that you're supposed to be there at certain times. There are usually a few people in a company who really have to, but the reason most employees work fixed hours is that the company can't measure their productivity.

The basic idea behind office hours is that if you can't make people work, you can at least prevent them from having fun. If employees have to be in the building a certain number of hours a day, and are forbidden to do non-work things while there, then they must be working. In theory. In practice they spend a lot of their time in a no-man's land, where they're neither working nor having fun.

If you could measure how much work people did, many companies wouldn't need any fixed workday. You could just say: this is what you have to do. Do it whenever you like, wherever you like. If your work requires you to talk to other people in the company, then you may need to be here a certain amount. Otherwise we don't care.

That may seem utopian, but it's what we told people who came to work for our company. There were no fixed office hours. I never showed up before 11 in the morning. But we weren't saying this to be benevolent. We were saying: if you work

here we expect you to get a lot done. Don't try to fool us just by being here a lot.

The problem with the facetime model is not just that it's demoralizing, but that the people pretending to work interrupt the ones actually working. I'm convinced the facetime model is the main reason large organizations have so many meetings. Per capita, large organizations accomplish very little. And yet all those people have to be on site at least eight hours a day. When so much time goes in one end and so little achievement comes out the other, something has to give. And meetings are the main mechanism for taking up the slack.

For one year I worked at a regular nine to five job, and I remember well the strange, cozy feeling that comes over one during meetings. I was very aware, because of the novelty, that I was being paid for programming. It seemed just amazing, as if there was a machine on my desk that spat out a dollar bill every two minutes no matter what I did. Even while I was in the bathroom! But because the imaginary machine was always running, I felt I always ought to be working. And so meetings felt wonderfully relaxing. They counted as work, just like programming, but they were so much easier. All you had to do was sit and look attentive.

Meetings are like an opiate with a network effect. So is email, on a smaller scale. And in addition to the direct cost in time, there's the cost in fragmentation-- breaking people's day up into bits too small to be useful.

You can see how dependent you've become on something by removing it suddenly. So for big companies I propose the following experiment. Set aside one day where meetings are forbidden-- where everyone has to sit at their desk all day and work without interruption on things they can do without talking to anyone else. Some amount of communication is necessary in most jobs, but I'm sure many employees could find eight hours worth of stuff they could do by themselves. You could call it "Work Day."

The other problem with pretend work is that it often looks better than real work. When I'm writing or hacking I spend as much time just thinking as I do actually typing. Half the time I'm sitting drinking a cup of tea, or walking around the neighborhood. This is a critical phase-- this is where ideas come from-- and yet I'd feel guilty doing this in most offices, with everyone else looking busy.

It's hard to see how bad some practice is till you have something to compare it to. And that's one reason open source, and even blogging in some cases, are so important. They show us what real work looks like.

We're funding eight new startups at the moment. A friend asked what they were doing for office space, and seemed surprised when I said we expected them to work out of whatever apartments they found to live in. But we didn't propose that to save money. We did it because we want their software to be good. Working in crappy informal spaces is one of the things startups do right without realizing it. As soon as you get into an office, work and life start to drift apart.

That is one of the key tenets of professionalism. Work and life are supposed to be separate. But that part, I'm convinced, is a mistake.

Bottom-Up

The third big lesson we can learn from open source and blogging is that ideas can bubble up from the bottom, instead of flowing down from the top. Open source and blogging both work bottom-up: people make what they want, and the best stuff prevails.

Does this sound familiar? It's the principle of a market economy. Ironically, though open source and blogs are done for free, those worlds resemble market economies, while most companies, for all their talk about the value of free markets, are run internally like communist states.

There are two forces that together steer design: ideas about what to do next, and the enforcement of quality. In the channel era, both flowed down from the top. For example, newspaper editors assigned stories to reporters, then edited what they wrote.

Open source and blogging show us things don't have to work that way. Ideas and even the enforcement of quality can flow bottom-up. And in both cases the results are not merely acceptable, but better. For example, open source software is more reliable precisely because it's open source; anyone can find mistakes.

The same happens with writing. As we got close to publication, I found I was very worried about the essays in Hackers & Painters that hadn't been online. Once an essay has had a couple thousand page views I feel reasonably confident about it. But these had had literally orders of magnitude less scrutiny. It felt like releasing software without testing it.

That's what all publishing used to be like. If you got ten people to read a manuscript, you were lucky. But I'd become so used to publishing online that the old method now seemed alarmingly unreliable, like navigating by dead reckoning once you'd gotten used to a GPS.

The other thing I like about publishing online is that you can write what you want and publish when you want. Earlier this year I wrote something that seemed suitable for a magazine, so I sent it to an editor I know. As I was waiting to hear back, I found to my surprise that I was hoping they'd reject it. Then I could put it online right away. If they accepted it, it wouldn't be read by anyone for months, and in the meantime I'd have to fight word-by-word to save it from being mangled by some twenty five year old copy editor. [5]

Many employees would *like* to build great things for the companies they work for, but more often than not management won't let them. How many of us have heard stories of employees going to management and saying, please let us build this thing to make money for you-- and the company saying no? The most famous

example is probably Steve Wozniak, who originally wanted to build microcomputers for his then-employer, HP. And they turned him down. On the blunderometer, this episode ranks with IBM accepting a non-exclusive license for DOS. But I think this happens all the time. We just don't hear about it usually, because to prove yourself right you have to quit and start your own company, like Wozniak did.

Startups

So these, I think, are the three big lessons open source and blogging have to teach business: (1) that people work harder on stuff they like, (2) that the standard office environment is very unproductive, and (3) that bottom-up often works better than top-down.

I can imagine managers at this point saying: what is this guy talking about? What good does it do me to know that my programmers would be more productive working at home on their own projects? I need their asses in here working on version 3.2 of our software, or we're never going to make the release date.

And it's true, the benefit that specific manager could derive from the forces I've described is near zero. When I say business can learn from open source, I don't mean any specific business can. I mean business can learn about new conditions the same way a gene pool does. I'm not claiming companies can get smarter, just that dumb ones will die.

So what will business look like when it has assimilated the lessons of open source and blogging? I think the big obstacle preventing us from seeing the future of business is the assumption that people working for you have to be employees. But think about what's going on underneath: the company has some money, and they pay it to the employee in the hope that he'll make something worth more than they paid him. Well, there are other ways to arrange that relationship. Instead of paying the guy money as a salary, why not give it to him as investment? Then instead of coming to your office to work on your projects, he can work wherever he wants on projects of his own.

Because few of us know any alternative, we have no idea how much better we could do than the traditional employer-employee relationship. Such customs evolve with glacial slowness. Our employer-employee relationship still retains a big chunk of master-servant DNA. [6]

I dislike being on either end of it. I'll work my ass off for a customer, but I resent being told what to do by a boss. And being a boss is also horribly frustrating; half the time it's easier just to do stuff yourself than to get someone else to do it for you. I'd rather do almost anything than give or receive a performance review.

On top of its unpromising origins, employment has accumulated a lot of cruft over the years. The list of what you can't ask in job interviews is now so long that for convenience I assume it's infinite. Within the office you now have to walk on eggshells lest anyone say or do something that makes the company prey to a

lawsuit. And God help you if you fire anyone.

Nothing shows more clearly that employment is not an ordinary economic relationship than companies being sued for firing people. In any purely economic relationship you're free to do what you want. If you want to stop buying steel pipe from one supplier and start buying it from another, you don't have to explain why. No one can accuse you of *unjustly* switching pipe suppliers. Justice implies some kind of paternal obligation that isn't there in transactions between equals.

Most of the legal restrictions on employers are intended to protect employees. But you can't have action without an equal and opposite reaction. You can't expect employers to have some kind of paternal responsibility toward employees without putting employees in the position of children. And that seems a bad road to go down.

Next time you're in a moderately large city, drop by the main post office and watch the body language of the people working there. They have the same sullen resentment as children made to do something they don't want to. Their union has exacted pay increases and work restrictions that would have been the envy of previous generations of postal workers, and yet they don't seem any happier for it. It's demoralizing to be on the receiving end of a paternalistic relationship, no matter how cozy the terms. Just ask any teenager.

I see the disadvantages of the employer-employee relationship because I've been on both sides of a better one: the investor-founder relationship. I wouldn't claim it's painless. When I was running a startup, the thought of our investors used to keep me up at night. And now that I'm an [investor,](#) the thought of our startups keeps me up at night. All the pain of whatever problem you're trying to solve is still there. But the pain hurts less when it isn't mixed with resentment.

I had the misfortune to participate in what amounted to a controlled experiment to prove that. After Yahoo bought our startup I went to work for them. I was doing exactly the same work, except with bosses. And to my horror I started acting like a child. The situation pushed buttons I'd forgotten I had.

The big advantage of investment over employment, as the examples of open source and blogging suggest, is that people working on projects of their own are enormously more productive. And a [startup](#) is a project of one's own in two senses, both of them important: it's creatively one's own, and also economically ones's own.

Google is a rare example of a big company in tune with the forces I've described. They've tried hard to make their offices less sterile than the usual cube farm. They give employees who do great work large grants of stock to simulate the rewards of a startup. They even let hackers spend 20% of their time on their own projects.

Why not let people spend 100% of their time on their own projects, and instead of trying to approximate the value of what they create, give them the actual market

value? Impossible? That is in fact what venture capitalists do.

So am I claiming that no one is going to be an employee anymore-- that everyone should go and start a startup? Of course not. But more people could do it than do it now. At the moment, even the smartest students leave school thinking they have to get a job. Actually what they need to do is make something valuable. A job is one way to do that, but the more ambitious ones will ordinarily be better off taking money from an investor than an employer.

Hackers tend to think business is for MBAs. But business administration is not what you're doing in a startup. What you're doing is business *creation*. And the first phase of that is mostly product creation-- that is, hacking. That's the hard part. It's a lot harder to create something people love than to take something people love and figure out how to make money from it.

Another thing that keeps people away from starting startups is the risk. Someone with kids and a mortgage should think twice before doing it. But most young hackers have neither.

And as the example of open source and blogging suggests, you'll enjoy it more, even if you fail. You'll be working on your own thing, instead of going to some office and doing what you're told. There may be more pain in your own company, but it won't hurt as much.

That may be the greatest effect, in the long run, of the forces underlying open source and blogging: finally ditching the old paternalistic employer-employee relationship, and replacing it with a purely economic one, between equals.

Notes

[1] Survey by Forrester Research reported in the cover story of Business Week, 31 Jan 2005. Apparently someone believed you have to replace the actual server in order to switch the operating system.

[2] It derives from the late Latin *tripalium*, a torture device so called because it consisted of three stakes. I don't know how the stakes were used. "Travel" has the same root.

[3] It would be much bigger news, in that sense, if the president faced unscripted questions by giving a press conference.

[4] One measure of the incompetence of newspapers is that so many still make you register to read stories. I have yet to find a blog that tried that.

[5] They accepted the article, but I took so long to send them the final version that by the time I did the section of the magazine they'd accepted it for had

disappeared in a reorganization.

[6] The word "boss" is derived from the Dutch *baas*, meaning "master."

Thanks to Sarah Harlin, Jessica Livingston, and Robert Morris for reading drafts of this.

■

[French Translation](#)

■

[Russian Translation](#)

■

[Japanese Translation](#)

■

[Spanish Translation](#)

■

[Arabic Translation](#)

After the Ladder

August 2005

Thirty years ago, one was supposed to work one's way up the corporate ladder. That's less the rule now. Our generation wants to get paid up front. Instead of developing a product for some big company in the expectation of getting job security in return, we develop the product ourselves, in a startup, and sell it to the big company. At the very least we want options.

Among other things, this shift has created the appearance of a rapid increase in economic inequality. But really the two cases are not as different as they look in economic statistics.

Economic statistics are misleading because they ignore the value of safe jobs. An easy job from which one can't be fired is worth money; exchanging the two is one of the commonest forms of corruption. A sinecure is, in effect, an annuity. Except sinecures don't appear in economic statistics. If they did, it would be clear that in practice socialist countries have nontrivial disparities of wealth, because they usually have a class of powerful bureaucrats who are paid mostly by seniority and can never be fired.

While not a sinecure, a position on the corporate ladder was genuinely valuable, because big companies tried not to fire people, and promoted from within based largely on seniority. A position on the corporate ladder had a value analogous to the "goodwill" that is a very real element in the valuation of companies. It meant one could expect future high paying jobs.

One of main causes of the decay of the corporate ladder is the trend for takeovers that began in the 1980s. Why waste your time climbing a ladder that might disappear before you reach the top?

And, by no coincidence, the corporate ladder was one of the reasons the early corporate raiders were so successful. It's not only economic statistics that ignore the value of safe jobs. Corporate balance sheets do too. One reason it was profitable to carve up 1980s companies and sell them for parts was that they hadn't formally acknowledged their implicit debt to employees who had done good work and expected to be rewarded with high-paying executive jobs when their time came.

In the movie *Wall Street*, Gordon Gekko ridicules a company overloaded with vice presidents. But the company may not be as corrupt as it seems; those VPs' cushy jobs were probably payment for work done earlier.

I like the new model better. For one thing, it seems a bad plan to treat jobs as rewards. Plenty of good engineers got made into bad managers that way. And the old system meant people had to deal with a lot more corporate politics, in order to protect the work they'd invested in a position on the ladder.

The big disadvantage of the new system is that it involves more risk. If you develop ideas in a startup instead of within a big company, any number of random factors could sink you before you can finish. But maybe the older generation would laugh at me for saying that the way we do things is riskier. After all, projects within big companies were always getting cancelled as a result of arbitrary decisions from higher up. My father's entire industry (breeder reactors) disappeared that way.

For better or worse, the idea of the corporate ladder is probably gone for good. The new model seems more liquid, and more efficient. But it is less of a change, financially, than one might think. Our fathers weren't *that* stupid.

■

Romanian Translation

■

Japanese Translation

THE END