

BLOG

TO BOOK



JEFF ATWOOD

Effective Programming: More Than Writing Code

Lean Software Development Advice for New
and Experienced Coders



www.codinghorror.com

Table of Contents

I.

Introduction

So You Want to Be a Programmer

The Eight Levels of Programmers

How to Write Without Writing

II.

The Art of Getting Shit Done

The Vast and Endless Sea

Sharpening the Saw

Go That Way, Really Fast

The Multi-Tasking Myth

III. Principles of Good Programming

The First Rule of Programming: It's Always Your Fault

The Best Code is No Code At All

Coding without Comments

Learn to Read the Source, Luke

Rubber Duck Problem Solving

Cultivate Teams, Not Ideas

Can Your Team Pass the Elevator Test?

Performance is a Feature

IV. Hiring Programmers the Right Way

Why Can't Programmers.. Program?

How to Hire a Programmer

Getting the Interview Phone Screen Right

The Years of Experience Myth

On Interviewing Programmers

Hardest Interview Puzzle Question Ever

V. Getting Your Team to Work Together

No Matter What They Tell You, It's a People Problem

Leading By Example

Vampires Programmers versus Werewolves Sysadmins

Pair Programming versus Code Review

Meetings: Where Work Goes to Die

Dealing With Bad Apples

The Bad Apple: Group Poison

On Working Remotely

VI. Your Batcave: Effective Workspaces for Programmers

The Programmer's Bill of Rights

Computer Workstation Ergonomics

Does More Than One Monitor Improve Productivity?

Investing in a Quality Programming Chair

Bias Lighting

VII. Designing With the User in Mind

You'll Never Have Enough Cheese

This is All Your App is: A Collection of Tiny Details

The User Interface is the Application

UI-First Software Development

The End of Pagination

Treating User Myopia

Revisiting The Fold

Fitts' Law and Infinite Width

The Ultimate Unit Test Failure

Version 1 Sucks, But Ship it Anyway

VIII. Security Basics: Protecting Your Users' Data

Should All Web Traffic Be Encrypted?

Dictionary Attacks 101

Speed Hashing

The Dirty Truth About Web Passwords

IX. Testing Your Code, So it Doesn't Suck More Than it Has To

Sharing the Customer's Pain

Working With the Chaos Monkey

Code Reviews: Just Do It

Testing With The Force

I Pity the Fool Who Doesn't Write Unit Tests

Unit Testing versus Beta Testing

Low-Fi Usability Testing

What's Worse Than Crashing?

X. Building, Managing and Benefiting from a Community

Listen To Your Community, But Don't Let Them Tell You What to Do

I Repeat: Do Not Listen to Your Users

The Gamification

Suspension, Ban or Hellban?

XI. Marketing Weasels and How Not

to Be One

9 Ways Marketing Weasels Will Try to Manipulate You

How Not to Advertise on the Internet

Groundhog Day, or, the Problem With A/B Testing

If it Looks Corporate, Change It

Software Pricing: Are We Doing it Wrong?

XII. Keeping Your Priorities Straight

Buying Happiness

Lived Fast, Died Young, Left a Tired Corpse

I.

Introduction



So You Want to Be a Programmer

“Not every programmer aspires to the same things in their career. But it’s illuminating to consider what a programmer could accomplish in ten years, twenty years, or thirty years — perhaps even a lifetime.”

I’d argue that the people who *need* to learn to code will be spurred on most of all by honesty, not religious faith in the truthiness of code as a universal good. Go in knowing both sides of the story, because [there are no silver bullets in code](#). If, after hearing both the pros and cons, you still want to learn to code, then by all means *learn to code*. If you’re so easily dissuaded by hearing a few downsides to coding, there are plenty of other things you could spend your time learning that are more unambiguously useful and practical. Per Michael Lopp, you could [learn to be a better communicator](#). Per Gina Trapani, you could [learn how to propose better solutions](#). Slinging code is [just a tiny part of the overall solution](#) in my experience. Why optimize for *that*?

On the earliest computers, everyone *had* to be a programmer because there was no software. If you wanted the computer to do anything, you wrote code. Computers in the not so distant past booted directly to the [friendly blinking cursor of a BASIC interpreter](#). I view the entire arc of software development as a field where we programmers spend our lives writing code so that our fellow human beings no longer *need* to write code (or even worse, become programmers) to get things done with computers. So this idea that “everyone must know how to code” is, to me, going backwards.



I fully support a push for basic Internet literacy. But in order to be a competent driver, does everyone need to know, in detail, how their automobile works? Must we teach all human beings the basics of being an auto mechanic, and elevate shop class to the same level as English and Mathematics classes? Isn't knowing how to change a tire, and when to take your car in for an oil change, sufficient? If your toilet is clogged, you shouldn't need to take a two week in depth plumbing course on toiletcademy.com to understand how to fix that. Reading a single web page, [just in time](#), should be more than adequate.

What is code, in the most abstract sense?

code (kōd) ...

1. *A system of signals used to represent letters or numbers in transmitting messages.*
2. *A system of symbols, letters, or words given certain arbitrary meanings, used for transmitting messages requiring secrecy or brevity.*
3. *A system of symbols and rules used to represent instructions to a computer...*

— *The American Heritage Dictionary of the English Language*

Is it punchcards? Remote terminals? Emacs? Textmate? Eclipse? Visual Studio? C? Ruby? JavaScript? In the 1920s, it was considered important to learn how to use slide rules. In the 1960s, it was considered important to learn mechanical drawing. None of that matters today. I'm hesitant to recommend any particular approach to coding other than the fundamentals as outlined in [Code: The Hidden Language of Computer Hardware and Software](#), because I'm not sure we'll even recognize coding in the next 20 or 30 years. To kids today, perhaps coding will eventually resemble [Minecraft](#), or [building levels in Portal 2](#).

But everyone should try writing a *little* code, because it somehow sharpens the mind, right?

Maybe in the same abstract way that [reading the entire Encyclopedia Britannica from beginning to end does](#). Honestly, I'd prefer that people spend their time discovering what problems they love and find interesting, first, and researching the hell out of those problems. The toughest thing in life is not learning a bunch of potentially hypothetically useful stuff, but figuring out [what the heck it is you want to do](#). If said research and exploration leads to coding, then by all means learn to code with my blessing ... which is worth exactly what it sounds like, nothing.

So, no, I don't advocate learning to code for the sake of learning to code. What I advocate is **shamelessly following your joy**. For example, I received the following email once:

I am a 45-year-old attorney/C.P.A. attempting to abandon my solo law practice as soon as humanly possible and strike out in search of my next vocation. I am actually paying someone to help me do this and, as a first step in the "find yourself" process, I was told to look back over my long and winding career and identify those times in my professional life when I was doing something I truly enjoyed.

Coming of age as an accountant during the PC revolution (when I started my first "real" job at Arthur Andersen we were still billing clients to update depreciation schedules manually), I spend a lot of time learning how to make computers, printers, and software (VisiCalc anyone?) work. This quasi-technical aspect of my work reached its apex when I was hired as a healthcare financial analyst for a large hospital system. When I arrived for my first day of work in that job, I learned that my predecessor had bequeathed me only a one page static Excel spreadsheet that purported to "analyze" a multi-million dollar managed care contract for a seven hospital health system. I proceeded to build my own

spreadsheet but quickly exceeded the database functional capacity of Excel and had to teach myself Access and thereafter proceeded to stretch the envelope of Access' spreadsheet capabilities to their utmost capacity – I had to retrieve hundreds of thousands of patient records and then perform pro forma calculations on them to see if the proposed contracts would result in more or less payment given identical utilization.

I will be the first to admit that I was not coding in any professional sense of the word. I did manage to make Access do things that MS technical support told me it could not do but I was still simply using very basic commands to bend an existing application to my will. The one thing I do remember was being happy. I typed infinitely nested commands into formula cells for twelve to fourteen hours a day and was still disappointed when I had to stop.

My experience in building that monster and making it run was, to date, my most satisfying professional accomplishment, despite going on to later become CFO of another healthcare facility, a feat that should have fulfilled all of my professional ambitions at that time. More than just the work, however, was the group of like-minded analysts and IT folks with whom I became associated as I tried, failed, tried, debugged, and continued building this behemoth of a database. I learned about Easter Eggs and coding lore and found myself hacking into areas of the hospital mainframe which were completely off-limits to someone of my paygrade. And yet, I kept pursuing my “professional goals” and ended up in jobs/careers I hated doing work I loathed.

Here's a person who a) found an interesting problem, b) attempted to create a solution to the problem, which naturally c) led him to learning to code. *And he loved it.* This is how it's supposed to work. I didn't become a programmer because someone told me learning to code was important, I became a programmer because [I wanted to change the rules of the video games I was playing](#), and learning to code was the only way to do that. Along the way, [I too fell in love](#).

All that to say that as I stand at the crossroads once more, I still hear the siren song of those halcyon days of quasi-coding during which I enjoyed my work. My question for you is whether you think it is even possible for someone of my vintage to learn to code to a level that I could be hired as a programmer. I am not trying to do this on the side while running the city of New York as a day job. Rather, I sincerely and completely want to become a bona fide programmer and spend my days creating (and/or debugging) something of value.

Unfortunately, [calling yourself a “programmer” can be a career-limiting move](#), particularly for someone who was a CFO in a previous career. People who work with

money tend to make a lot of money; [see Wall Street](#).

But this isn't about money, is it? [It's about love](#). **So, if you want to be a programmer, all you need to do is follow your joy and fall in love with code.** Any programmer worth their salt immediately recognizes a fellow true believer, a person as madly in love with code as they are, warts and all. [Welcome to the tribe](#).

And if you're reading this and thinking, "screw this Jeff Atwood guy, who is he to tell me whether I should learn to code or not", all I can say is: [good! That's the spirit!](#)

The Eight Levels of Programmers

Have you ever gotten that classic job interview question, “**where do you see yourself in five years?**” When asked, I’m always mentally transported back to [a certain Twisted Sister video from 1984](#).

I want you to tell me — no, better yet, stand up and tell the class —



what do you wanna do with your life?

You want to rock, naturally! Or at least be a [rockstar programmer](#). It’s not a question that typically gets a serious answer — sort of like that other old groan-inducing interview chestnut, “what’s your greatest weakness?” It’s that you sometimes rock *too hard*, right? Innocent bystanders could get hurt.

But I think this is a different and more serious class of question, one that deserves real consideration. Not for the interviewer’s benefit, but for your *own* benefit.

The “where do you see yourself in five years” question is sort of glib, and most people have a pat answer they give to interviewers. But it does raise some deeper concerns: what *is* the potential career path for a software developer? Sure, [we do this stuff because](#)

[we love it](#), and we're [very fortunate in that regard](#). But will you be sitting in front of your computer programming when you're 50? When you're 60? What is the best possible career outcome for a programmer who aspires to be.. well, a programmer?

What if I told you, with [tongue firmly planted in cheek](#), that there were **Eight Levels of Programmers?**

1. **Dead Programmer**

This is the highest level. Your code has survived and transcended your death. You are a part of the permanent historical record of computing. Other programmers study your work and writing. You may have won a Turing Award, or written influential papers, or invented one or more pieces of fundamental technology that have affected the course of programming as we know it. You don't just have a wikipedia entry — there are entire websites dedicated to studying your life and work.

Very few programmers ever achieve this level in their own lifetimes.

Examples: [Dijkstra](#), [Knuth](#), [Kay](#)

2. **Successful Programmer**

Programmers who are both well known and have created entire businesses — perhaps even whole industries — around their code. These programmers have given themselves [the real freedom zero](#): the freedom to decide for themselves what they want to work on. And to share that freedom with their fellow programmers.

This is the level to which most programmers should aspire. Getting to this level often depends more on business skills than programming.

Examples: [Gates](#), [Carmack](#), [DHH](#)

3. **Famous Programmer**

This is also a good place to be, but not unless you also have a day job.

You're famous in programming circles. But being famous doesn't necessarily mean you can turn a profit and support yourself. Famous is good, but *successful* is better. You probably work for a large, well-known technology company, an influential small company, or you're a part of a modest startup team. Either way, other programmers have heard of you, and you're having a positive impact on the field.

4. **Working Programmer**

You have a successful career as a software developer. Your skills are always in demand and you never have to look very long or hard to find a great job. Your peers respect you. Every company you work with is improved and enriched in some way by your presence.

But where do you go from there?

5. **Average Programmer**

At this level you are a good enough programmer to realize that you're not a *great* programmer. And you might never be.

Talent often has little to do with success. You can be very successful if you have business and people skills. If you are an average programmer but manage to make a living at it then you *are* talented, just not necessarily at coding.

Don't knock the value of self-awareness. It's more rare than you realize. There's nothing wrong with lacking talent. Be bold. Figure out what you're good at, and pursue it. Aggressively.

6. **Amateur Programmer**

An amateur programmer loves to code, and it shows: they might be a promising student or intern, or perhaps they're contributing to open source projects, or building interesting "just for fun" applications or websites in their spare time. Their code and ideas show promise and enthusiasm.

Being an amateur is a good thing; from this level one can rapidly rise to become a working programmer.

7. **Unknown Programmer**

The proverbial typical programmer. Joe Coder. Competent (usually) but unremarkable. Probably works for a large, anonymous MegaCorp. It's just a job, not their entire life. Nothing wrong with that, either.

8. **Bad Programmer**

People who somehow fell into the programmer role without an iota of skill or ability. Everything they touch [turns into pain and suffering](#) for their fellow programmers — with the possible exception of *other* Bad Programmers, who lack even the rudimentary skill required to tell that they're working with another Bad Programmer.

Which is, perhaps, the hallmark of all Bad Programmers. These people have no business writing code of any kind — but they do, anyway.

These levels aren't entirely serious. Not every programmer aspires to the same things in their career. But it's illuminating to consider what a programmer *could* accomplish in ten years, twenty years, or thirty years — perhaps even a lifetime. Which [notable programmers](#) do you admire the most? What did they accomplish to earn your admiration?

In short, **what do you wanna do with your life?**

How to Write Without Writing

I have a confession to make: in a way, I founded [Stack Overflow](#) to **trick my fellow programmers**.

Before you trot out the pitchforks and torches, let me explain.

Over the last six years, I've come to believe deeply in the idea that becoming a great programmer has very little to do with *programming*. Yes, it takes a modicum of technical skill and dogged persistence, absolutely. But even more than that, it takes [serious communication skills](#):

The difference between a tolerable programmer and a great programmer is not how many programming languages they know, and it's not whether they prefer Python or Java. It's whether they can communicate their ideas. By persuading other people, they get leverage. By writing clear comments and technical specs, they let other programmers understand their code, which means other programmers can use and work with their code instead of rewriting it. Absent this, their code is worthless.

That is of course a quote from my co-founder Joel Spolsky, and it's one of my favorites.

In defense of my fellow programmers, communication with other human beings is not exactly what we signed up for. We didn't launch our careers in software development because we loved chatting with folks. Communication is just plain *hard*, particularly written communication. How exactly do you get better at something you self-selected out of? [Blogging is one way](#):

People spend their entire lives learning how to write effectively. It isn't something you can fake. It isn't something you can buy. You have to work at it.

That's exactly why people who are afraid they can't write should be blogging.

It's exercise. No matter how out of shape you are, if you exercise a few times a week, you're bound to get fitter. Write a small blog entry a few times every week and you're bound to become a better writer. If you're not writing because you're intimidated by writing, well, you're likely to stay that way forever.

Even with the best of intentions, telling someone “you should blog!” never works. I know this from painful first hand experience. Blogging isn’t for everyone. Even a small blog entry can seem like an insurmountable, impenetrable, arbitrary chunk of writing to the average programmer. How do I get my fellow programmers to blog without blogging, to write without *writing*?

By cheating like *hell*, that’s how.

Consider [this letter I received](#):

I’m not sure if you have thought about this side effect or not, but Stack Overflow has taught me more about writing effectively than any class I’ve taken, book I’ve read, or any other experience I have had before.

I can think of no other medium where I can test my writing chops (by writing an answer), get immediate feedback on its quality (particularly when writing quality trumps technical correctness, such as subjective questions) and see other peoples’ attempts as well and how they compare with mine. Votes don’t lie and it gives me a good indicator of how well an email I might send out to future co-workers would be received or a business proposal I might write.

Over the course of the past 5 months all the answers I’ve been writing have been more and more refined in terms of the quality. If I don’t end up as the top answer I look at the answer that did and study what they did differently and where I faltered. Was I too verbose or was I too terse? Was I missing the crux of the question or did I hit it dead on?

I know that you said that writing your Coding Horror blog helped you greatly in refining your writing over the years. Stack Overflow has been doing the same for me and I just wanted to thank you for the opportunity. I’ve decided to setup a coding blog in your footsteps and I just registered a domain today. Hopefully that will go as well as writing on SO has. There are no tougher critics than fellow programmers who scrutinize every detail, every technical remark and grammar structure looking for mistakes. If you can effectively write for and be accepted by a group of programmers you can write for anyone.

Joel and I have always positioned Stack Overflow, and all the other [Stack Exchange Q&A sites](#), as lightweight, focused, “fun size” units of writing.

Yes, by God, **we will trick you into becoming a better writer if that’s what it takes - and it always does**. Stack Overflow has many overtly game-like elements, but it is a game in service of the greater good - to make the Internet better, and more

importantly, to make *you* better. Seeing my fellow programmers naturally improve their written communication skills while participating in a focused, expert Q&A community with their peers? Nothing makes me prouder.

Beyond programming, there's a whole other community of peers out there who grok how important writing is, and will support you in [sharpening your saw, er, pen](#). We have our own, too.



If you're an **author, editor, reviewer, blogger, copywriter or aspiring writer of any kind, professional or otherwise** — check out writers.stackexchange.com.

Becoming a more effective writer is the one bedrock skill that will further your professional career, no matter *what* you choose to do.

But mostly, you should write. I thought Jon Skeet [summed it up particularly well here](#):

Everyone should write a lot — whether it's a blog, a book, Stack Overflow answers, emails or whatever. Write, and take some care over it. Clarifying your communication helps you to clarify your own internal thought processes, in my experience. It's amazing how much you find you don't know when you try to explain something in detail to someone else. It can start a whole new process of discovery.

The process of writing is indeed a journey of discovery, one that will last the rest of your life. It doesn't ultimately matter whether you're writing a novel, a printer review, a Stack Overflow answer, fan fiction, a blog entry, a comment, a technical whitepaper, some emo LiveJournal entry, or even meta-talk about writing itself. Just **get out there and write!**

Gift this book to your friends...for [free](#).

Become a Hyperink reader. Get a [special surprise](#).

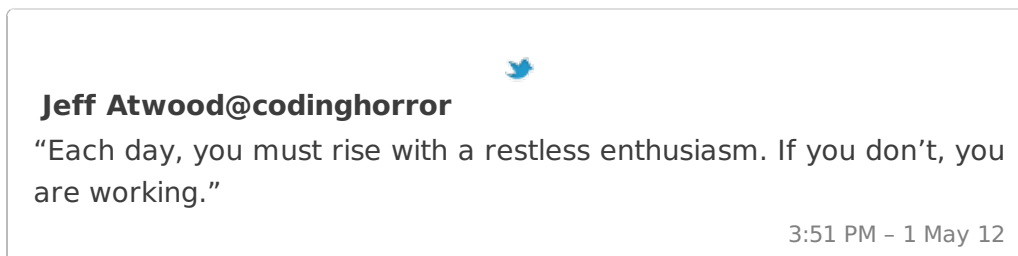
Like the book? Support our author and leave a [comment!](#)

II.

The Art of Getting Shit Done



The Vast and Endless Sea



After we created Stack Overflow, some people were convinced we had built a marginally better mousetrap for asking and answering questions. The inevitable speculation began: **can we use your engine to build a Q&A site about {topic}?** Our answer was Stack Exchange. Pay us \$129 a month (and up), and you too can create a hosted Q&A community on our engine — for whatever topic you like!

Well, I have a confession to make: my heart was never in Stack Exchange. It was a parallel effort in a parallel universe only tangentially related to my own. There’s a whole host of reasons why, but if I had to summarize it in a sentence, I’d say that **money is poisonous to communities**. That \$129/month doesn’t sound like much — and it isn’t — but the commercial nature of the enterprise permeated and distorted everything from the get-go.

Yes, Stack Overflow Internet Services Incorporated©®™ is technically a business, even [a venture-capital backed business](#) now — but I didn’t co-found it because I wanted to make money. I co-founded it because **I wanted to build something cool that made the internet better**. Yes, selfishly for myself, of course, but also in conjunction with all of my fellow programmers, because I know [none of us is as dumb as all of us](#).

Nobody is participating in Stack Overflow to *make money*. We’re participating in Stack Overflow because ...

- We love programming.
- We want to leave breadcrumb trails for other programmers to follow so they can avoid making the same dumb mistakes we did.
- Teaching peers is one of the best ways to develop mastery.

- We can follow our own interests wherever they lead.
- We want to collectively build something great for the community with our tiny slices of effort.

I don't care how much you pay me, you'll never be able to recreate the incredibly satisfying feeling I get when **demonstrating mastery within my community of peers**. That's what we do on Stack Overflow: have *fun*, while making the internet one infinitesimally tiny bit better every day.

So is it any wonder that some claim [Stack Overflow is more satisfying than their real jobs?](#) Not to me.

If this all seems like a bunch of **communist hippie bullcrap** to you, I understand. It's hard to explain. But there is quite a bit of science documenting these strange motivations. Let's start with Dan Pink's 2009 TED talk.



WATCH: [Daniel Pink|TED Talk| 2009](#)

Dan's talk centers on [the candle problem](#). Given the following three items ...

1. A candle
2. A box of thumbtacks
3. A book of matches

... how can you attach the candle to the wall?

It's not a very interesting problem on its own — that is, until you try to **incentivize** teams to solve it:

Now I want to tell you about an experiment using the candle problem by a scientist from

Princeton named Sam Glucksberg. Here's what he did.

To the first group, he said, "I'm going to time you to establish norms, averages for how long it typically takes someone to solve this sort of problem."

To the second group, he said, "If you're in the top 25 percent of the fastest times you get five dollars. If you're the fastest of everyone we're testing here today you get 20 dollars." (This was many years ago. Adjusted for inflation, it's a decent sum of money for a few minutes of work.)

Question: How much faster did this group solve the problem?

Answer: It took them, on average, three and a half minutes longer. Three and a half minutes longer. Now this makes no sense, right? I mean, I'm an American. I believe in free markets. That's not how it's supposed to work. If you want people to perform better, you reward them. Give them bonuses, commissions, their own reality show. Incentivize them. That's how business works. But that's not happening here. You've got a monetary incentive designed to sharpen thinking and accelerate creativity – and it does just the opposite. It dulls thinking and blocks creativity.

It turns out that traditional carrot-and-stick incentives are only useful for repetitive, mechanical tasks. The minute you have to do anything even slightly complex that requires even a little problem solving without a clear solution or rules — those incentives not only don't work, they *make things worse!*

Pink eventually wrote a book about this, [Drive: The Surprising Truth About What Motivates Us](#).

There's no need to read the book; this clever ten-minute whiteboard animation will walk you through the main points. If you view only one video today, view this one.



WATCH: [RSA Daniel Pink Drive](#)|[RSA Animate](#)

The concept of [intrinsic motivation](#) may not be a new one, but I find that very few companies are brave enough to actually implement them.

I've tried mightily to live up to the ideals that Stack Overflow was founded on when building out my team. I don't care when you come to work or what your schedule is. I don't care [where in the world you live](#) (provided you have a great internet connection). I don't care how you do the work. I'm not going to micromanage you and assign you a queue of task items. There's no need.

If you want to build a ship, don't drum up the men to gather wood, divide the work and give orders. Instead, teach them to yearn for the vast and endless sea.

- [Antoine de Saint-Exupéry](#)

Because I know you yearn for the vast and endless sea, just like we do.

Sharpening the Saw

As a software developer, how do you **sharpen your saw**?



Sharpening the saw is shorthand for anything you do that isn't *programming*, necessarily, but (theoretically) makes you a better programmer. It's derived from the Covey book [The 7 Habits of Highly Effective People](#).

There's a guy who stumbled into a lumberjack in the mountains. The man stops to observe the lumberjack, watching him feverishly sawing at this very large tree. He noticed that the lumberjack was working up a sweat, sawing and sawing, yet going nowhere. The bystander noticed that the saw the lumberjack was using was about as sharp as a butter knife. So, he says to the lumberjack, "Excuse me Mr. Lumberjack, but I couldn't help noticing how hard you are working on that tree, but going nowhere." The lumberjack replies with sweat dripping off of his brow, "Yes... I know. This tree seems to be giving me some trouble." The bystander replies and says, "But Mr. Lumberjack, your saw is so dull that it couldn't possibly cut through anything." "I know", says the lumberjack, "but I am too busy sawing to take time to sharpen my saw."

Of course, the best way to improve at something is to [do it as often as possible](#). But if

you're so heads down in coding that you have no time for discussion, introspection or study, you aren't really moving forward. You have to strike a mindful balance between practicing your craft and thinking about how you practice your craft.

Scott Hanselman has some solid ideas on ways to [encourage members of your development team to sharpen their saws](#). And then there's the obvious way, the thing you're doing right now: **reading programming blogs**. If you keep an open mind, you can sharpen your saw that way, [as Reginald Braithwaite notes](#):

What we do is this: we read a blog post, reading thing after thing we agree with, and if just one thing in there doesn't fit our personal world view, we demand a correction. If the thesis of the post clashes with our prejudices, we accuse the author of being an idiot. Honestly, we would suck as salespeople. We would quit the first time someone disagreed with us.

What I suggest we do is mimic these salespeople. When we read a post, or a book, or look at a new language, let's assume that some or even most of it will not be new. Let's assume that we'll positively detest some of it. But let's also look at it in terms of our own profit: we win if we can find just one thing in there that makes us better programmers.

That's all we need from a blog post, you know. It's a huge win if there's one thing in a post. Heck, it's a huge win if we read one hundred posts and learn one new valuable thing.

If you're looking for good programming blogs to sharpen your saw (or at least pique your intellectual curiosity), I know of two **excellent programming specific link aggregation sites** that can help you find them.

The first is [Hacker News](#), which I recommend highly.

Y Hacker News new | comments | leaders | jobs | submit login

1. ▲ **How to be a program manager** (joelonsoftware.com)
109 points by twampss 16 hours ago | 35 comments
2. ▲ **Be Relentlessly Resourceful** (paulgraham.com)
202 points by herdrick 1 day ago | 126 comments
3. ▲ **An open letter to Steve Jobs about approving the AMBER alert application** (zdzierski.com)
12 points by jgrahamc 3 hours ago | 5 comments
4. ▲ **Tarsnap reaches profitability** (daemonology.net)
81 points by cperciva 17 hours ago | 47 comments
5. ▲ **TED: Mike Rowe talks about dirty jobs and innovation** (ted.com)
61 points by geuis 14 hours ago | 11 comments
6. ▲ **Color Scheme Designer 3** (colorschemedesigner.com)
89 points by melvinram 21 hours ago | 17 comments
7. ▲ **There's always time to launch your dream** (37signals.com)
10 points by screwperman 2 hours ago | 3 comments
8. ▲ **TheSixtyOne (YC W09) Is Building a Digg For Indie Music** (techcrunch.com)
47 points by markbao 13 hours ago | 18 comments
9. ▲ **Voxli (YC W09) Targets Gamers With Browser-Based Group Voice Chat** (techcrunch.com)
45 points by andrewow 14 hours ago | 28 comments
10. ▲ **"I Quit" 1 Year anniversary** (emadibrahim.com)
82 points by ebrahim 22 hours ago | 43 comments

[Hacker News](#) is the brainchild of [Paul Graham](#), so it partially reflects his interests in [Y Combinator](#) and entrepreneurial stuff like startups. Paul is serious about moderation on the site, so in addition to the typical Digg-style voting, there's a secret cabal (I like to think of it as [The Octagon](#), "no one will admit they still exist!") of hand-picked editors who remove flagged posts. More importantly, the conversation on the site about the articles is quite rational, with very little noise and trolling.

The other site is [programming reddit](#). The conversation there is more chaotic, with a wild-west, anything-goes sensibility, gated only by the up and down votes of the community. But it is quite reliable for digging up a great variety of links that are of particular interest to programmers.

Of course, too much saw sharpening, or random, aimless saw sharpening, can become [another form of procrastination](#). But a developer who seems completely disinterested in it at all is a huge red flag. As Peter Bregman explains, [obsession can be a good thing](#):

People are often successful not despite their dysfunctions but because of them. Obsessions are one of the greatest telltale signs of success. Understand a person's obsessions and you will understand her natural motivation. The thing for which she would walk to the end of the earth.

It's OK to be a *little* obsessed with sharpening your saw, if it means actively **submitting and discussing programming articles on, say, [Hacker News](#)**.

What do you recommend for sharpening *your* saw as a programmer?

Go That Way, Really Fast

When it comes to running Stack Overflow, [the company](#), I take all my business advice from one person, and one person alone: **Curtis Armstrong**.



More specifically, Curtis Armstrong as **Charles De Mar** from the 1985 absurdist teen comedy classic, [Better Off Dead](#). When asked for advice on how to ski down a particularly treacherous mountain, he replied:



Go that way, really fast. If something gets in your way ... turn.

In the five months since we [announced our funding](#), we have ...

- Built an [international team](#)
- Created an entirely new open, democratic process for creating Q&A sites at [Area 51](#)
- Launched ~24 new community-driven [Stack Exchange network sites](#)
- Implemented [per-site meta discussion](#) and [per-site real time chat](#)
- Rolled out new versions of [Careers](#) and [Jobs](#)
- Built and open-sourced a tool for exploring and sharing all our creative commons data in the [Stack Exchange Data Explorer](#)
- Finalized V1 of the [Stack Exchange API](#), for building your own apps against our Q&A platform

... and honestly, I'm a little worried we're *still not going fast enough*.

There are any number of Stack Overflow engine clones out there already, and I say more power to 'em. I'm proud to have something worth copying. If we do nothing else except help lead the world away from the ancient, creaky, horribly broken bulletin board model of phpBB and vBulletin — attempting to get information out of those things is like **panning for gold in a neverending river of sewage** — then that is more than I could have ever hoped for.

It is our stated goal as a company to live in harmony with the web, by only doing things that we believe make the internet better, at least in some small way. No, seriously. It's in writing and everything, I swear! We're not here to subvert or own anyone or anything. We just love community, and we love getting great answers to our questions. So if something gets in our way while doing that, well, we're not gonna fight you. **We'll just turn.** And keep going forward, really fast. Which is why those clones better move quick if they want to keep up with us.

While I like to think that having Charles De Mar as a business advisor is unique to our company, the idea that speed is important is hardly original to us. For example, certain Google projects also appear to understand [Boyd's Law of Iteration](#).

Boyd decided that the primary determinant to winning dogfights was not observing, orienting, planning or acting better. The primary determinant to winning dogfights was observing, orienting, planning and acting faster. In other words, how quickly one could iterate. Speed of iteration, Boyd suggested, beats quality of iteration.

Speed of iteration — the [Google Chrome](#) project has it.

1.0	December 11, 2008
2.0	May 24, 2009
3.0	October 12, 2009
4.0	January 25, 2010
5.0	May 25, 2010
6.0	September 2, 2010

Chrome was a completely respectable browser in V1 and V2. The entire project has moved forward so fast that it now is, at least in my humble opinion, the best browser on the planet. Google went from nothing, no web browser at all, to best-of-breed in **under two years**. Meanwhile, Internet Explorer took longer than the entire development period of Chrome to go from version 7 to version 8. And by the time Internet Explorer 9 ships — even though it's actually looking like Microsoft's best, most competent technical upgrade of the browser yet — it will be completely outclassed at launch by both Firefox and Chrome.

The [Google Android](#) project is another example. Android doesn't have to be better than the iPhone (and it most *definitely* isn't; it's been mediocre at best until recent versions). They just need to be *faster at improving*. Google is pushing out [Froyos and Gingerbreads and Honeycombs](#) with incredible, breakneck speed. Yes, Apple has indisputably better

taste — and an impeccably controlled experience. But at their current rate of progress, they'll be playing second or third fiddle to Google in the mobile space inside a few years. It's inevitable.

So, until further notice, we'll be following the same strategy as the Android and Chrome teams. **We're going to go that way, really fast. And if something gets in our way, we'll turn.**

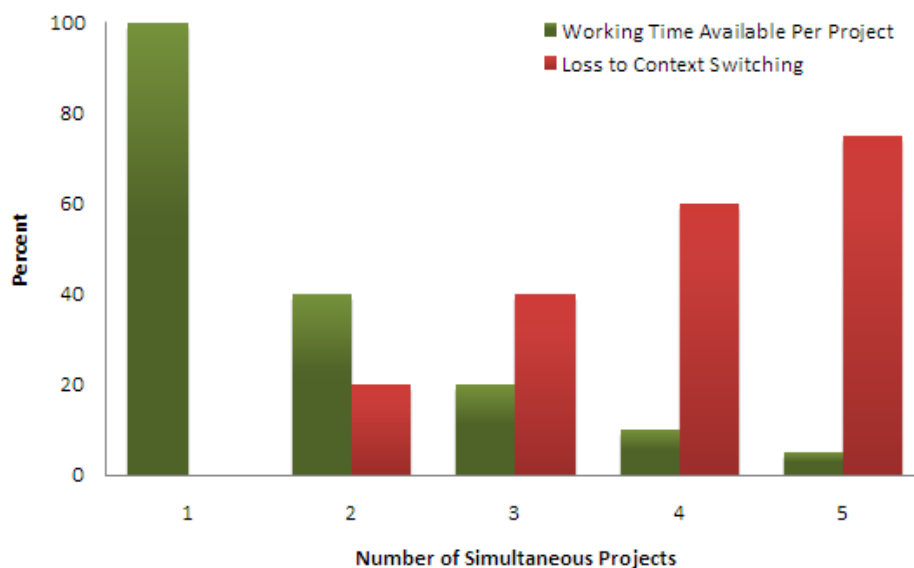
 **Tim O'Reilly@timoreilly**

"Larry Page responds"high correlation between speed and good decisions...There are good fast decisions but no good slow decisions"

3:40 PM - 27 Sep 11

The Multi-Tasking Myth

In [Quality Software Management: Systems Thinking](#), Gerald Weinberg proposed a rule of thumb to calculate the waste caused by project switching:



Even adding a single project to your workload is profoundly debilitating by Weinberg's calculation. You lose 20 percent of your time. By the time you add a third project to the mix, nearly half your time is wasted in task switching.

This can be a problem even if you're only working on a single project at any time. The impact of simply letting your email, phone and instant messaging interrupt what you're doing can be profound, as documented in this [BBC study](#):

The study, carried out at the Institute of Psychiatry, found excessive use of technology reduced workers' intelligence. Those distracted by incoming email and phone calls saw a 10-point fall in their IQ — more than twice that found in studies of the impact of smoking marijuana, said researchers.

Kathy Sierra wrote a great [post comparing multi-tasking and serial tasks](#) and followed it up a year later with a typically insightful post proposing that [multi-tasking makes us stupid](#):

Perhaps the biggest problem of all, though, is that the majority of people doing the most media multitasking have a big-ass blind spot on just how much they suck at it.

We believe we can e-mail and talk on the phone at the same time, with little or no degradation of either communication.

We believe we can do homework while watching a movie.

We believe we can surf the web while talking to our kids/spouse/lover/co-worker.

But we can't! Not without a hit on every level — time, quality and the ability to think deeply.

Joel Spolsky [compares the task switching penalty for computers and computer programmers](#):

The trick here is that when you manage programmers, specifically, task switches take a really, really, really long time. That's because programming is the kind of task where you have to keep a lot of things in your head at once. The more things you remember at once, the more productive you are at programming. A programmer coding at full throttle is keeping zillions of things in their head at once: everything from names of variables, data structures, important APIs, the names of utility functions that they wrote and call a lot, even the name of the subdirectory where they store their source code. If you send that programmer to Crete for a three week vacation, they will forget it all. The human brain seems to move it out of short-term RAM and swaps it out onto a backup tape where it takes forever to retrieve.

I've often pushed back on demands to work on multiple projects at the same time. It can be [difficult to say no](#), because software developers are notoriously prone to the [occupational hazard of optimism](#).

We typically [overestimate how much we'll actually get done](#), and multi-tasking exaggerates our own internal biases even more. Whenever possible, avoid interruptions and avoid working on more than one project at the same time. If it's unavoidable, **be brutally honest with yourself — and your stakeholders — about how much you can actually get done under multi-tasking conditions.** It's probably less than you think.

 **Merlin Mann@hotdogsladies**

“Good thing you’re tagging all those “Low Priority” tasks. God forbid you’d ever lose track of shit that’s not worth doing.”

12:43 PM - 1 Feb 12

Gift this book to your friends...for [free](#).

Become a Hyperink reader. Get a [special surprise](#).

Like the book? Support our author and leave a [comment](#)!

III.

Principles of Good Programming



The First Rule of Programming: It's Always Your Fault

 **Jeff Atwood@codinghorror**

“We should endeavor to fix ourselves before accusing the world of being broken.”

12:22 PM - 30 May 12

You know the feeling. It's happened to all of us at some point: you've pored over the code a dozen times and *still* can't find a problem with it. But there's some bug or error you can't seem to get rid of. There just has to be something wrong with the machine you're coding on, with the operating system you're running under, with the tools and libraries you're using. There just *has to be!*

No matter how desperate you get, don't choose that path. Down that path lies voodoo computing and [programming by coincidence](#). In short, madness.

It's frustrating to repeatedly bang your head against difficult, obscure bugs, but don't let desperation lead you astray. An essential part of [being a humble programmer](#) is realizing that whenever there's a problem with the code you've written, **it's always your fault**. This is aptly summarized in [The Pragmatic Programmer](#) as “Select Isn't Broken”:

In most projects, the code you are debugging may be a mixture of application code written by you and others on your project team, third-party products (database, connectivity, graphical libraries, specialized communications or algorithms, and so on) and the platform environment (operating system, system libraries, and compilers).

It is possible that a bug exists in the OS, the compiler, or a third-party product- but this should not be your first thought. It is much more likely that the bug exists in the application code under development. It is generally more profitable to assume that the application code is incorrectly calling into a library than to assume that the library itself is broken. Even if the problem does lie with a third party, you'll still have to eliminate your code before submitting the bug report.

We worked on a project where a senior engineer was convinced that the select system call was broken on Solaris. No amount of persuasion or logic could change his mind (the fact that every other networking application on the box worked fine was irrelevant). He spent weeks writing workarounds, which, for some odd reason, didn't seem to fix the problem. When finally forced to sit down and read the documentation on select, he discovered the problem and corrected it in a matter of minutes. We now use the phrase "select is broken" as a gentle reminder whenever one of us starts blaming the system for a fault that is likely to be our own.

The flip side of [code ownership](#) is *code responsibility*. No matter what the problem is with your software — maybe it's not even your code in the first place — **always assume the problem is in your code** and act accordingly. If you're going to subject the world to your software, take full responsibility for its failures. Even if, technically speaking, you don't have to. That's how you earn respect and credibility. You certainly don't earn respect or credibility by endlessly pawning off errors and problems on other people, other companies, other sources.

Statistically, you understand, it is incredibly rare for any bugs or errors in your software *not* to be your fault. In [Code Complete](#), Steve McConnell cited two studies that proved it:

A pair of studies performed [in 1973 and 1984] found that, of total errors reported, roughly 95% are caused by programmers, 2% by systems software (the compiler and the operating system), 2% by some other software, and 1% by the hardware. Systems software and development tools are used by many more people today than they were in the 1970s and 1980s, and so my best guess is that, today, an even higher percentage of errors are the programmers' fault.

Whatever the problem with your software is, take ownership. Start with your code, and investigate further and further outward until you have definitive evidence of where the problem lies. If the problem lies in some other bit of code that you don't control, you'll not only have learned essential troubleshooting and diagnostic skills, you'll also have an audit trail of evidence to back up your claims, too. This is certainly a lot more work than shrugging your shoulders and pointing your finger at the OS, the tools, or the framework — but it also engenders a sense of trust and respect you're unlikely to achieve through fingerpointing and evasion.

If you truly aspire to being a humble programmer, you should have no qualms about saying "hey, this is my fault — and I'll get to the bottom of it."

The Best Code is No Code At All

 **Jeff Atwood@codinghorror**

“you can never have too little minimalism”

11:29 AM - 21 May 12

Rich Skrenta writes that [code is our enemy](#):

Code is bad. It rots. It requires periodic maintenance. It has bugs that need to be found. New features mean old code has to be adapted. The more code you have, the more places there are for bugs to hide. The longer checkouts or compiles take. The longer it takes a new employee to make sense of your system. If you have to refactor there's more stuff to move around.

Code is produced by engineers. To make more code requires more engineers. Engineers have n^2 communication costs, and all that code they add to the system, while expanding its capability, also increases a whole basket of costs. You should do whatever possible to increase the productivity of individual programmers in terms of the expressive power of the code they write. Less code to do the same thing (and possibly better). Less programmers to hire. Less organizational communication costs.

Rich hints at it here, but the real problem isn't the code. The code, like a newborn babe, is blameless and innocent the minute it is written into the world. Code isn't our enemy. You want to see the real enemy? Go look in the mirror. There's your problem, right there.



As a software developer, you are your own worst enemy. The sooner you realize that, the better off you'll be.

I know you have the best of intentions. We all do. We're software developers; we love writing code. It's what we do. We never met a problem we couldn't solve with some duct tape, a jury-rigged coat hanger and a pinch of code. But Wil Shipley argues that we should [rein in our natural tendencies to write lots of code](#):

The fundamental nature of coding is that our task, as programmers, is to recognize that every decision we make is a trade-off. To be a master programmer is to understand the nature of these trade-offs, and be conscious of them in everything we write.

In coding, you have many dimensions in which you can rate code:

- *Brevity of code*

- *Featurefulness*
- *Speed of execution*
- *Time spent coding*
- *Robustness*
- *Flexibility*

Now, remember, these dimensions are all in opposition to one another. You can spend three days writing a routine which is really beautiful and fast, so you've gotten two of your dimensions up, but you've spent three days, so the "time spent coding" dimension is way down.

So, when is this worth it? How do we make these decisions? The answer turns out to be very sane, very simple, and also the one nobody, ever, listens to: Start with brevity. Increase the other dimensions as required by testing.

I couldn't agree more. I've given similar advice when I [exhorted developers to Code Smaller](#). And I'm not talking about a [reductio ad absurdum](#) contest where we use up all the clever tricks in our books to make the code fit into less physical space. **I'm talking about practical, sensible strategies to reduce the volume of code an individual programmer has to read to understand how a program works.** Here's a trivial little example of what I'm talking about:

```
if (s == String.Empty)if (s == "")
```

It seems obvious to me that the latter case is better because it's just plain *smaller*. And yet I'm virtually guaranteed to encounter developers who will fight me, almost [literally to the death](#), because they're absolutely convinced that the verbosity of `String.Empty` is somehow friendlier to the compiler. As if I care about that. As if *anyone* cared about that!

It's painful for most software developers to acknowledge this, because they love code so much, but **the best code is no code at all**. Every new line of code you willingly bring into the world is code that has to be debugged, code that has to be read and understood, code that has to be supported. Every time you write new code, you should do so reluctantly, under duress, because you completely exhausted all your other options. Code is only our enemy because there are so many of us programmers writing so damn much of it. If you *can't* get away with no code, the next best thing is to **start with brevity**.

If you love writing code — really, truly *love to write code* — you'll love it enough to write

as little of it as possible.

Coding without Comments

If peppering your code with lots of comments is good, then having **zillions of comments** in your code must be *great*, right? Not quite. Excess is one way [good comments go bad](#):

```

*****
' Name: CopyString
' Purpose: This routine copies a string from the source
' string (source) to the target string (target).
'
' Algorithm: It gets the length of "source" and then copies each
' character, one at a time, into "target". It uses
' the loop index as an array index into both "source"
' and "target" and increments the loop/array index
' after each character is copied.
'
' Inputs: input The string to be copied
'
' Outputs: output The string to receive the copy of "input"
'
' Interface Assumptions: None
'
' Modification History: None
'
' Author: Dwight K. Coder
' Date Created: 10/1/04
' Phone: (555) 222-2255
' SSN: 111-22-3333
' Eye Color: Green
' Maiden Name: None
' Blood Type: AB-
' Mother's Maiden Name: None
' Favorite Car: Pontiac Aztek
' Personalized License Plate: "Tek-ie"
*****

```

I'm constantly running across comments from developers who don't seem to understand that the code already tells us *how* it works; [we need the comments to tell us why it works](#). Code comments are so widely misunderstood and abused that you might find yourself wondering if they're worth using at all. Be careful what you wish for. Here's some code with **no comments whatsoever**:

```

r = n / 2;
while ( abs( r - (n/r) ) > t ) {
    r = 0.5 * ( r + (n/r) );
}
System.out.println( "r = " + r );

```

Any idea what that bit of code does? It's perfectly readable, but *what the heck does it do?*

Let's add a comment.

```

// square root of n with Newton-Raphson approximation
r = n / 2;
while ( abs( r - (n/r) ) > t ) {
    r = 0.5 * ( r + (n/r) );
}
System.out.println( "r = " + r );

```

That must be what I was getting at, right? Some sort of pleasant, middle-of-the-road compromise between the two polar extremes of no comments whatsoever and carefully formatted epic poems every second line of code?

Not exactly. Rather than add a comment, I'd refactor to this:

```
private double SquareRootApproximation(n) {
    r = n / 2;
    while ( abs( r - (n/r) ) > t ) {
        r = 0.5 * ( r + (n/r) );
    }
    return r;
}
System.out.println( "r = " + SquareRootApproximation(r) );
```

I haven't added a single comment, and yet this mysterious bit of code is now perfectly understandable.

While comments are neither inherently good or bad, they are frequently used as a crutch. **You should always write your code as if comments didn't exist.** This *forces* you to write your code in the simplest, plainest, most self-documenting way you can humanly come up with.

When you've rewritten, refactored, and rearchitected your code a dozen times to make it easy for your fellow developers to read and understand — when you can't possibly imagine any conceivable way your code could be changed to become more straightforward and obvious — then, and *only then*, should you feel compelled to add a comment explaining what your code does.

As [Steve points out](#), this is one key difference between junior and senior developers:

In the old days, seeing too much code at once quite frankly exceeded my complexity threshold, and when I had to work with it I'd typically try to rewrite it or at least comment it heavily. Today, however, I just slog through it without complaining (much). When I have a specific goal in mind and a complicated piece of code to write, I spend my time making it happen rather than telling myself stories about it [in comments].

Junior developers rely on comments to tell the story when they should be relying on the *code* to tell the story. Comments are narrative asides; important in their own way, but in no way meant to replace plot, characterization, and setting.

Perhaps that's the dirty little secret of code comments: **to write good comments you have to be a good writer.** Comments aren't code meant for the compiler, they're words meant to communicate ideas to other human beings. While I do (mostly) love my fellow programmers, I can't say that effective communication with other human beings is exactly our strong suit. I've seen three-paragraph emails from developers on my teams that practically melted my brain. *These* are the people we're trusting to write clear, understandable comments in our code? I think maybe some of us might be better off sticking to our strengths — that is, writing for the compiler, in as clear a way as we

possibly can, and reaching for the comments only as a method of last resort.

Writing good, meaningful comments is hard. It's as much an art as writing the code itself; maybe even more so. As Sammy Larbi said in [Common Excuses Used To Comment Code](#), if you feel your code is too complex to understand *without* comments, **your code is probably just bad**. Rewrite it until it doesn't need comments any more. If, at the end of that effort, you still feel comments are necessary, then by all means, add comments. Carefully.

Learn to Read the Source, Luke

In the calculus of communication, writing coherent paragraphs that your fellow human beings can comprehend and understand is [far more difficult](#) than tapping out a few lines of software code that the interpreter or compiler won't barf on.

That's why, when it comes to code, [all the documentation probably sucks](#). And because writing for people is way harder than writing for machines, the documentation will *continue* to suck for the foreseeable future. There's very little you can do about it.

Except for one thing.



You can **learn to read the source, Luke**.

The [transformative power of "source always included" in JavaScript](#) is a major reason why I coined — and continue to believe in — [Atwood's Law](#). Even if "view source" isn't built in (but it totally should be), you should demand access to the underlying source code for your stack. **No matter what the documentation says, the source code is the ultimate truth, the best and most definitive and up-to-date documentation you're likely to find.** This will be true *forever*, so the sooner you come to terms with this, the better off you'll be as a software developer.

I had a whole entry I was going to write about this, and then I discovered [Brandon Bloom's](#) brilliant [post](#) on the topic at Hacker News. Read closely, because he explains the virtue of reading source, and in what context you *need* to read the source, far better than I could:

I started working with Microsoft platforms professionally at age 15 or so. I worked for Microsoft as a software developer doing integration work on Visual Studio. More than ten years after I first wrote a line of Visual Basic, I wish I could never link against a closed library ever again.

Using software is different than building software. When you're using most software for its primary function, it's a well worn path. Others have encountered the problems and enough people have spoken up to prompt the core contributors to correct the issue. But when you're building software, you're doing something new. And there are so many ways to do it, you'll encounter unused bits, rusty corners, and unfinished experimental code paths. You'll encounter edge cases that have been known to be broken, but were worked around.

Sometimes, the documentation isn't complete. Sometimes, it's wrong. The source code never lies. For an experienced developer, reading the source can often be faster... especially if you're already familiar with the package's architecture. I'm in a medium-sized co-working space with several startups. A lot of the other CTOs and engineers come to our team for guidance and advice on occasion. When people report a problem with their stack, the first question I ask them is: "Well, did you read the source code?"

I encourage developers to git clone anything and everything they depend on. Initially, they are all afraid. "That project is too big, I'll never find it!" or "I'm not smart enough to understand it" or "That code is so ugly! I can't stand to look at it". But you don't have to search the whole thing, you just need to follow the trail. And if you can't understand the platform below you, how can you understand your own software? And most of the time, what inexperienced developers consider beautiful is superficial, and what they consider ugly, is battle-hardened production-ready code from master hackers. Now, a year or two later, I've had a couple of developers come up to me and thank me for forcing them to sink or swim in other people's code bases. They are better at their craft and they wonder how they ever got anything done without the source code in the past.

When you run a business, if your software has a bug, your customers don't care if it is your fault or Linus' or some random Rails developer's. They care that your software is bugged. Everyone's software becomes my software because all of their bugs are my bugs. When something goes wrong, you need to seek out what is broken, and you need

to fix it. You fix it at the right spot in the stack to minimize risks, maintenance costs, and turnaround time. Sometimes, a quick workaround is best. Other times, you'll need to recompile your compiler. Often, you can ask someone else to fix it upstream, but just as often, you'll need to fix it yourself.

- Closed-software shops have two choices: beg for generosity, or work around it.
- Open source shops with weaker developers tend to act the same as closed-software shops.
- Older shops tend to slowly build the muscles required to maintain their own forks and patches and whatnot.

True hackers have come to terms with a simple fact: *If it runs on my machine, it's my software. I'm responsible for it. I must understand it. Building from source is the rule and not an exception. I must control my environment and I must control my dependencies.*

Nobody reads other people's code for fun. Hell, [I don't even like reading my own code](#). The idea that you'd settle down in a deep leather chair with your smoking jacket and a snifter of brandy for a fine evening of reading through someone else's code is absurd.

But we need access to the source code. We *must* read other people's code [because we have to understand it to get things done](#). So don't be afraid to **read the source, Luke** — and follow it wherever it takes you, no matter how scary looking that code is.

Rubber Duck Problem Solving

At [Stack Exchange](#), we insist that people who ask questions *put some effort into their question, and we're kind of jerks about it*. That is, when you set out to ask a question, you should ...

- Describe what's happening in sufficient detail that we can follow along. Provide the necessary background for us to understand what's going on, even if we aren't experts in your particular area.
- Tell us why you *need* to know the answer. What led you here? Is it idle curiosity or somehow blocking you on a project? We don't require your whole life story, just give us some basic context for the problem.
- Share any research you did toward solving your problem, and what you found, if anything. And if you didn't do any research — should you even be asking?
- Ultimately, this is about fairness: if you're going to ask us to spend our valuable time helping you, it's only fair that you put in a reasonable amount of your valuable time into crafting a decent question. Help us help you!

We have a great [How to Ask page](#) that explains all of this, which is linked generously throughout the network. (And on Stack Overflow, due to massive question volume, we actually *force* new users to click through that page before asking their first question. You can see this yourself by clicking on Ask Question in incognito or anonymous browser mode.)

What we're trying to prevent, most of all, is the unanswerable drive-by question. Those help nobody, and left unchecked they can ruin a Q&A site, turning it into a virtual ghost town. On Stack Exchange, questions that are so devoid of information and context that they can't reasonably be answered will be actively closed, and if they aren't improved, eventually deleted.

As I said, we're kinda jerks about this rule. But for good reason: we're not-so-subtly trying to **help you help yourself**, by teaching you [Rubber Duck problem solving](#). And boy does it ever work.



It's quite common. See for yourself:

[How can I thank the community when I solve my own problems?](#)

I've only posted one question so far, and almost posted another. In both cases, I answered my own questions at least partially while writing it out. I credit the community and the process itself for making me think about the answer. There's nothing explicit in what I'm writing that states quite obviously the answer I needed, but something about writing it down makes me think along extra lines of thought.

[Why is it that properly formulating your question often yields you your answer?](#)

I don't know how many times this has happened:

- I have a problem
- I decide to bring it to stack overflow
- I awkwardly write down my question
- I realize that the question doesn't make any sense
- I take 15 minutes to rethink how to ask my question
- I realize that I'm attacking the problem from a wrong direction entirely

- I start from scratch and find my solution quickly

Does this happen to you? Sometimes asking the right question seems like half the problem.

[Beginning to ask a question actually helps me debug my problem myself.](#)

Beginning to ask a question actually helps me debug my problem myself, especially while trying to formulate a coherent and detailed enough question body in order to get decent answers. Has this happened to anybody else before?

It's not a new concept, and every community seems to figure it out on their own given enough time, but "Ask the Duck" is [a very powerful problem-solving technique](#).

Bob pointed into a corner of the office. "Over there," he said, "is a duck. I want you to ask that duck your question."

I looked at the duck. It was, in fact, stuffed, and very dead. Even if it had not been dead, it probably would not have been a good source of design information. I looked at Bob. Bob was dead serious. He was also my superior, and I wanted to keep my job.

I awkwardly went to stand next to the duck and bent my head, as if in prayer, to commune with this duck. "What," Bob demanded, "are you doing?"

"I'm asking my question of the duck," I said.

One of Bob's superintendants was in his office. He was grinning like a bastard around his toothpick. "Andy," he said, "I don't want you to pray to the duck. I want you to ask the duck your question."

I licked my lips. "Out loud?" I said.

"Out loud," Bob said firmly.

I cleared my throat. "Duck," I began.

"It's name is Bob Junior," Bob's superintendent supplied. I shot him a dirty look.

"Duck," I continued, "I want to know, when you use a clevis hanger, what keeps the sprinkler pipe from jumping out of the clevis when the head discharges, causing the pipe to..."

In the middle of asking the duck my question, the answer hit me. The clevis hanger is

suspended from the structure above by a length of all-thread rod. If the pipe-fitter cuts the all-thread rod such that it butts up against the top of the pipe, it essentially will hold the pipe in the hanger and keep it from bucking.

I turned to look at Bob. Bob was nodding. "You know, don't you," he said.

"You run the all-thread rod to the top of the pipe," I said.

"That's right," said Bob. "Next time you have a question, I want you to come in here and ask the duck, not me. Ask it out loud. If you still don't know the answer, then you can ask me."

"Okay," I said, and got back to work.

I love this particular story because it makes it crystal clear how **the critical part of rubber duck problem solving is to totally commit to asking a thorough, detailed question of this imaginary person or inanimate object**. Yes, even if you end up throwing the question away because you eventually realize that you made some dumb mistake. The effort of walking an imaginary someone through your problem, step by step and in some detail, is what will often lead you to your answer. But if you *aren't* willing to put the effort into fully explaining the problem and how you've attacked it, you can't reap the benefits of thinking deeply about your own problem *before* you ask others to.

If you don't have a [coding buddy \(but you totally should\)](#), you can leverage the Rubber Duck problem solving technique to figure out problems all by yourself, or with the benefit of the greater Internet community. Even if you don't get the answer you wanted, forcing yourself to fully explain your problem — [ideally in writing](#) — will frequently lead to new insights and discoveries.

Cultivate Teams, Not Ideas

How much is a good idea worth? According to Derek Sivers, [not much](#):

It's so funny when I hear people being so protective of ideas. (People who want me to sign an NDA to tell me the simplest idea.) To me, ideas are worth nothing unless executed. They are just a multiplier. Execution is worth millions.

To make a business, you need to multiply the two. The most brilliant idea, with no execution, is worth \$20. The most brilliant idea takes great execution to be worth \$20,000,000. That's why I don't want to hear people's ideas. I'm not interested until I see their execution.

I was reminded of Mr. Sivers article when [this email](#) made the rounds earlier this month:

I feel that this story is important to tell you because Kickstarter.com copied us. I tried for 4 years to get people to take Fundable seriously, traveling across the country, even giving a presentation to FBFund, Facebook's fund to stimulate development of new apps. It was a series of rejections for 4 years. I really felt that I presented myself professionally in every business situation and I dressed appropriately and practiced my presentations. That was not enough. The idiots wanted us to show them charts with massive profits and widespread public acceptance so that they didn't have to take any risks.

All it took was 5 super-connected people at Kickstarter (especially Andy Baio) to take a concept we worked hard to refine, tweak it with Amazon Payments, and then take credit. You could say that that's capitalism, but I still think you should acknowledge people that you take inspiration from. I do. I owe the concept of Fundable to many things, including living in cooperative student housing and studying Political Science at Michigan. Rational choice theory, tragedy of the commons, and collective action are a few political science concepts that are relevant to Fundable.

Yes, Fundable had some technical and customer service problems. That's because we had no money to revise it. I had plans to scrap the entire CMS and start from scratch with a new design. We were just so burned out that motivation was hard to come by. What was the point if we weren't making enough money to live on after 4 years?

The disconnect between idea and execution here is so vast it's hard to understand why the author himself can't see it.

I wouldn't call ideas *worthless*, per se, but it's clear that ideas alone are a hollow sort of currency. Success is rarely determined by the quality of your ideas. But it *is* frequently determined by the quality of your execution. So instead of worrying about whether the Next Big Idea you're all working on is sufficiently brilliant, **worry about how well you're executing.**

The criticism that all you need is "super-connected people" to be successful was also leveled at Stack Overflow. In an email to me last year, Andy Baio — ironically, the very person being cited in the email — said:

I very much enjoyed [the Hacker News conversation about cloning the site in a weekend](#). My favorite comments were from the people that believe Stack Overflow is only successful because of the Cult of Atwood & Spolsky. Amazing.

I don't care how internet famous you are; *nobody* gets a pass on execution. Sure, you may have a few more eyeballs at the beginning, but if you don't build something useful, the world will eventually just shrug its collective shoulders and move along to more useful things.

In software development, execution is staying on top of all the tiny details that make up your app. If you're not constantly obsessing over every aspect of your application, relentlessly polishing and improving every little part of it — no matter how trivial — you're not executing. At least, not well.

And unless you work alone, which is a rarity these days, your ability to stay on top of the collection of tiny details that makes up your app will hinge entirely on whether or not you can build a great team. They are the building block of any successful endeavor. This talk by [Ed Catmull](#) is almost exclusively focused on how Pixar learned, through trial and error, to build teams that can *execute*.



WATCH: [Ed Catmull, Pixar|Keep Your Crises Small](#)

It's a fascinating talk, full of some great insights, and you should watch the whole thing. In it, Mr. Catmull amplifies Mr. Sivers' sentiment:

If you give a good idea to a mediocre group, they'll screw it up. If you give a mediocre idea to a good group, they'll fix it. Or they'll throw it away and come up with something else.

Execution isn't merely a multiplier. It's far more powerful. How your team executes has the power to transform your idea from gold into lead, or from lead into gold. That's why, when building Stack Overflow, I was so fortunate to not only [work with Joel Spolsky](#), but also to cherry-pick two of the best developers I had ever worked with in my previous jobs and drag them along with me — kicking and screaming if necessary.

If I had to point to **the one thing that made our project successful**, it was not the idea behind it, our internet fame, the tools we chose, or the funding we had (precious little, for the record).

It was our team.

The value of my advice is debatable. But you would do well to heed the advice of Mr. Sivers and Mr. Catmull. If you want to be successful, stop worrying about the great ideas, and concentrate on cultivating great teams.

Can Your Team Pass the Elevator Test?

Software developers do [love to code](#). But very few of them, in my experience, can explain *why* they're coding. Try this exercise on one of your teammates if you don't believe me. Ask them what they're doing. Then ask them why they're doing it, and keep asking until you get to a reason your customers would understand.

What are you working on?

I'm fixing the sort order on this datagrid.

Why are you working on that?

Because it's on the bug list.

Why is it on the bug list?

Because one of the testers reported it as a bug.

Why was it reported as a bug?

The tester thinks this field should sort in numeric order instead of alphanumeric order.

Why does the tester think that?

Evidently the users are having trouble finding things when item 2 is sorted under item 19.

If this conversation seems strange to you, you probably haven't worked with many software developers. Like [the number of licks it takes to get to the center of a tootsie pop](#), it might surprise you just how many times you have to ask "why" until you get to something — *anything* — your customers would actually care about.

It's a big disconnect.

Software developers think their job is writing code. But it's not.* Their job is to solve the customer's problem. Sure, our preferred medium for solving problems is software, and that does involve writing code. But let's keep this squarely in context:

writing code is something you *have* to do to deliver a solution. It is not an end in and of itself.

As software developers, we spend so much time mired in endless, fractal levels of detail that it's all too easy for us to fall into the trap of coding for the sake of coding. Without a clear focus and something to rally around, we lose the context around our code. That's why [it's so important to have a clear project vision statement](#) that everyone can use as a touchstone on the project. If you've got the vision statement down, **every person on your team should be able to pass the “elevator test” with a stranger — to clearly explain what they're working on, and why anyone would care, within 60 seconds.**

If your team can't explain their work to a layperson in a meaningful way, you're in trouble, whether you realize it or not. But you are in good company. Jim Highsmith is here to help. He explains [a quick formula for building a project vision model](#):

A product vision model helps team members pass the elevator test — the ability to explain the project to someone within two minutes. It comes from Geoffrey Moore's book [Crossing the Chasm](#). It follows the form:

for (target customer)

who (statement of need or opportunity)

the (product name) is a (product category)

that (key benefit, compelling reason to buy)

unlike (primary competitive alternative)

our product (statement of primary differentiation)

Creating a product vision statement helps teams remain focused on the critical aspects of the product, even when details are changing rapidly. It is very easy to get focused on the short-term issues associated with a 2-4 week development iteration and lose track of the overall product vision.

I'm not a big fan of formulas, because they're so, well, *formulaic*. But it's a reasonable starting point. Play [Mad Libs](#) and see what you come up with. It's worlds better than no vision statement, or an uninspiring, rambling, ad-hoc mess masquerading as a vision statement. However, I think Jim's second suggestion for developing a vision statement holds much more promise.

Even within an IT organization, I think every project should be considered to produce a “product.” Whether the project results involve enhancements to an internal accounting system or a new e-commerce site, product-oriented thinking pays back benefits.

One practice that I’ve found effective in getting teams to think about a product vision is the Design-the-Box exercise. This exercise is great to open up a session to initiate a project. **The team makes the assumption that the product will be sold in a shrink-wrapped box, and their task is to design the product box front and back.** This involves coming up with a product name, a graphic, three to four key bullet points on the front to “sell” the product, a detailed feature description on the back, and operating requirements.

Coming up with 15 or 20 product features proves to be easy. It’s figuring out which 3 or 4 would cause someone to buy the product that is difficult. One thing that usually happens is an intense discussion about who the customers really are.

Design-the-Box is a fantastic way to formulate a vision statement. It’s based on a concrete, real world concept that most people can easily wrap their heads around. Forget those pie-in-the-sky vision quests: **what would our (hypothetical) product box look like?**

We’re all consumers; the design goals for a product box are obvious and universal. What is a product box if not the ultimate elevator pitch? It should...

- Explain what our product is in the simplest possible way.
- Make it crystal clear why a potential customer would want to buy this product.
- Be uniquely identifiable amongst all the other boxes on the shelf.

Consider the box for the [ill-fated Microsoft Bob](#) product as an example. How do you explain why customers should want Microsoft Bob? How would you even explain what the heck Microsoft Bob *is*?

Microsoft Home

8 MB of memory (RAM) required

Requires 3.5" high density disk.

Write a letter to Mom.

Remember your brother's birthday.

Balance your budget.

Get the pet to the vet.

And much more!

E-mail friends in Phoenix.

Be more organized than ever before.

Microsoft BOB

Introducing Hard-working, Easy-going Software Everyone Will Use

Meet the BOB!

With personalized help unlike any you've ever experienced before and familiar everyday items to work with, Microsoft Bob® helps you get going right away.

Why will everyone in your household love BOB?

Because these eight essential household programs can help you keep your hectic household humming!

1. Calendar
Keep calendars, including the day, week, month, and year, with digital or individual calendars and To-Do Lists. You can even use the easy address book.

2. Letter Writer
Write a letter to your bank, utility, or even a friend, or create a new address book. It's easy to use and you can even use the easy address book.

3. Checkbook
Managing your check book, managing your budget, and tracking spending have never been easier.

4. Address Book
Never miss another birthday! It's a snap to keep track of addresses, birthdays, and other important information, and to create special address books. You can even make mailing lists automatically.

5. E-Mail
Now it's easy for your whole household to stay on track with each other and the world around the world.

6. Financial Guide
Prepare the tax returns for your household in minutes with the easy-to-use software. It's so easy to use, you can even make mailing lists automatically.

7. Household Manager
Be more organized than ever before! Manage your household with the easy-to-use software. It's so easy to use, you can even make mailing lists automatically.

8. Greetings
Experience a world of exciting cards for all occasions. Personalize your cards with a message for the occasion.

To use Microsoft Bob, you need:

- Personal computer with a 486 or higher processor
- MS-DOS operating system version 3.2 or later
- Microsoft Windows operating system version 1.0 or later or Microsoft Windows 95
- 8 MB of memory (RAM) and 32 MB of available hard disk space
- 15" high-density disk drive
- Super VGA 256-color monitor
- Microsoft Mouse or compatible pointing device
- Mouse to use 1.0 (compatibility subscription fee apply)
- Mouse to use the Checkbook Plus On-line feature, which is available in the United States only (monthly subscription fee apply)
- Optional: Audio board with speakers or headphones

Note: This package contains CD-ROM high-density disks and information on ordering a CD-ROM.

Microsoft Bob is available in English, Spanish, and French. It is not available in other languages. Microsoft Bob is a registered trademark of Microsoft Corporation. © 1997 Microsoft Corporation. All rights reserved.

Microsoft Bob is available in English, Spanish, and French. It is not available in other languages. Microsoft Bob is a registered trademark of Microsoft Corporation. © 1997 Microsoft Corporation. All rights reserved.

Make yourself at home. Customize your "Home" with different screens, styles, and fun objects. Keep your files private—or share information with others. Even start programs for MS-DOS and Windows right from Bob.

Friends of Bob. Choose from more than 100 Friends of Bob who are happy to assist you when and where you need it—they'll even learn how you like to work!

These eight easy-going, hard-working programs are built. So you can add entries from the Address Book to letters automatically. And in your Calendar and To-Do List, you can get reminders of tasks from your Household Manager, the dates from your Checkbook, or birthdays from your Address Book. Bob works hard to keep things simple for you!

Microsoft Home
© 1997 Microsoft Corporation. All rights reserved.
0295 Part No. 64035

Microsoft
Satisfaction Guarantee
Great Greetings

It's instructive to look at existing product boxes you find effective, and those you find ineffective. We definitely know [what our product box shouldn't look like](#).

Have a [rock solid vision statement for your project from day one](#). If you don't, use one of Jim's excellent suggestions to build one up immediately. **Without a coherent vision statement, it's appalling how many teams can't pass the elevator test — they can't explain what it is they're working on, or why it matters.** Don't make that same mistake. Get a kick-ass vision statement that your teammates can relate their work to. Make sure *your* team can pass the elevator test.

* Completely stolen from Billy Hollis' great 15-minute software addicts talk.

 **Jeff Atwood@codinghorror**

“When a code addict needs a fix, they just do a few extra lines.”

3:26 AM - 15 May 12

Performance is a Feature

We've always put a heavy emphasis on performance at Stack Overflow and [Stack Exchange](#). Not just because we're performance wonks (guilty!), but because we think speed is a competitive advantage. There's [plenty of experimental data](#) proving that **the slower your website loads and displays, the less people will use it.**

[Google found that] the page with 10 results took 0.4 seconds to generate. The page with 30 results took 0.9 seconds. Half a second delay caused a 20 percent drop in traffic. Half a second delay killed user satisfaction.

In A/B tests, [Amazon] tried delaying the page in increments of 100 milliseconds and found that even very small delays would result in substantial and costly drops in revenue.

I believe the converse of this is also true. That is, the faster your website is, the *more* people will use it. This follows logically if you think like [an information omnivore](#): the faster you can load the page, the faster you can tell whether that page contains what you want. Therefore, you should always favor fast websites. The opportunity cost for switching on the public internet is effectively nil, and whatever it is that you're looking for, there are multiple websites that offer a similar experience. So how do you distinguish yourself? **You start by being, above all else, fast.**

Do you, too, [feel the need - the need for speed](#)? If so, I have three pieces of advice that I'd like to share with you.

1. Follow the Yahoo Guidelines. Religiously.

The golden reference standard for building a fast website remains [Yahoo's 13 Simple Rules for Speeding Up Your Web Site](#) from 2007. There is one caveat, however:

There's some good advice here, but there's also a lot of advice that only makes sense if you run a website that gets millions of unique users per day. Do you run a website like that? If so, what are you doing reading this instead of flying your private jet to a Bermuda vacation with your trophy wife?

So ... a funny thing happened to me since I wrote that four years ago. I now run [a network of public, community driven Q&A web sites](#) that *do* get millions of daily unique

users. (I'm still waiting on the jet and trophy wife.) It does depend a little on the size of your site, but if you run a public website, **you really should [pore over Yahoo's checklist](#) and take every line of it to heart.** Or use the tools that do this for you:

- [Yahoo YSlow](#)
- [Google Page Speed](#)
- [Pingdom Tools](#)

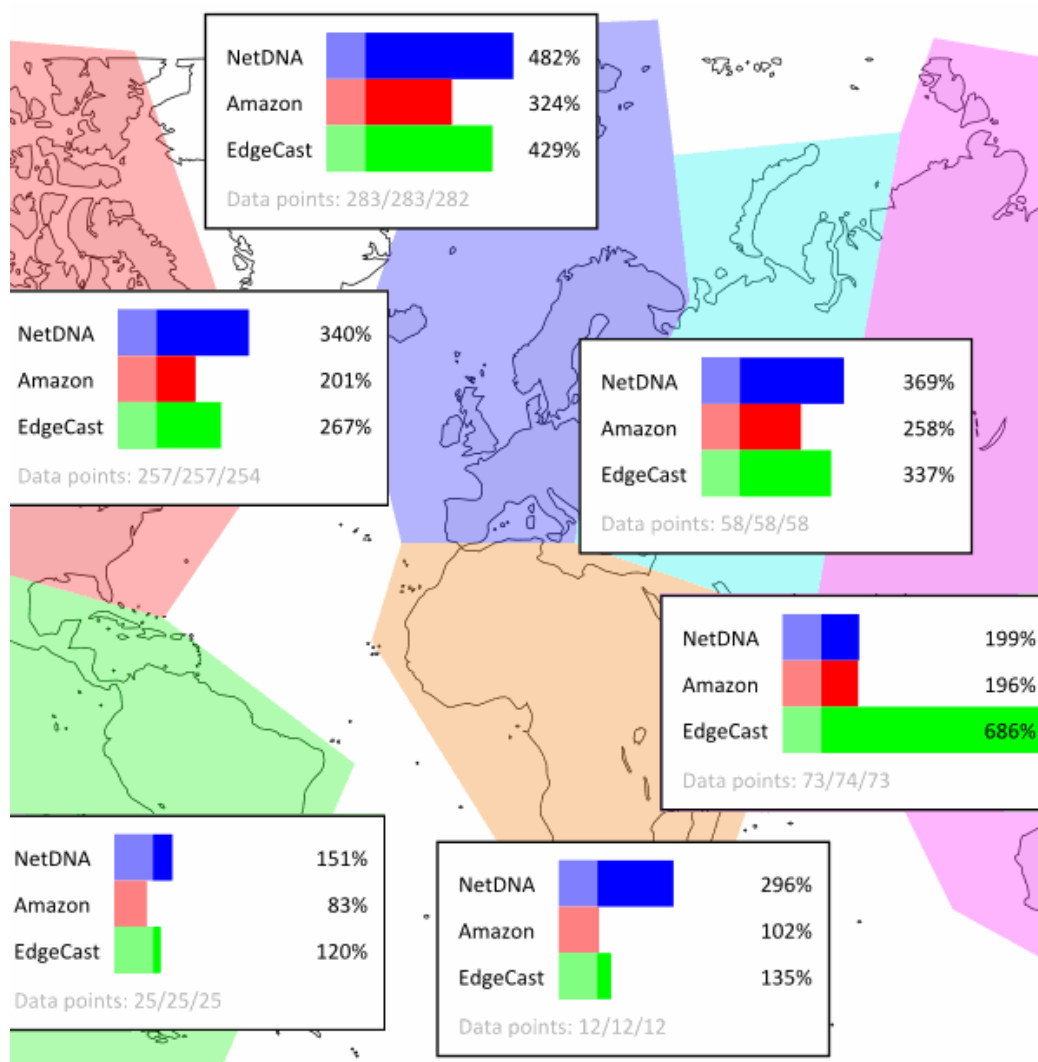
We've long since implemented most of the 13 items on Yahoo's list, except for one. But it's a big one: [Using a Content Delivery Network](#).

The user's proximity to your web server has an impact on response times. Deploying your content across multiple, geographically dispersed servers will make your pages load faster from the user's perspective. But where should you start?

As a first step to implementing geographically dispersed content, don't attempt to redesign your web application to work in a distributed architecture. Depending on the application, changing the architecture could include daunting tasks such as synchronizing session state and replicating database transactions across server locations. Attempts to reduce the distance between users and your content could be delayed by, or never pass, this application architecture step.

Remember that 80 to 90 percent of the end-user response time is spent downloading all the components in the page: images, stylesheets, scripts, Flash, etc. This is the *Performance Golden Rule*. Rather than starting with the difficult task of redesigning your application architecture, it's better to first disperse your static content. This not only achieves a bigger reduction in response times, but it's easier thanks to content delivery networks.

As a final optimization step, we just [rolled out a CDN for all our static content](#). The results are promising; the baseline here is our datacenter in NYC, so the below should be read as *"how much faster did our website get for users in this area of the world?"*



In the interests of technical accuracy, static content isn't the complete performance picture; you still have to talk to our servers in NYC to get the dynamic content which is the meat of the page. But 90 percent of our visitors are anonymous, only 36 percent of our traffic is from the USA, and Yahoo's research shows that [40 to 60 percent of daily visitors come in with an empty browser cache](#). Optimizing this cold cache performance worldwide is a *huge* win.

Now, I would not recommend going *directly* for a CDN. I'd leave that until later, as there are a bunch of performance tweaks on Yahoo's list which are free and trivial to implement. But using a CDN has gotten a heck of a lot less expensive and much simpler since 2007, with lots more competition in the space from companies like [Amazon](#), [NetDNA](#) and [CacheFly](#). So when the time comes, and you've worked through the Yahoo list as religiously as I recommend, you'll be ready.

2. Love (and Optimize for) Your Anonymous and Registered Users

Our Q&A sites are all about making the internet better. That’s why all the contributed content is [licensed back to the community under Creative Commons](#) and *always* visible regardless of whether you are logged in or not. I [despise walled gardens](#). In fact, you don’t actually have to log in *at all* to participate in Q&A with us. Not even a little!

The primary source of our traffic is [anonymous users arriving from search engines](#) and elsewhere. It’s classic “write once, read — and [hopefully edit](#) — millions of times.” But we are also making the site richer and more dynamic for our avid community members, who definitely *are* logged in. We add features all the time, which means we’re serving up more JavaScript and HTML. There’s an unavoidable tension here between the download footprint for users who are on the site every day, and users who may visit once a month or once a year.

Both classes are important, but have fundamentally different needs. Anonymous users are voracious consumers optimizing for rapid browsing, while our avid community members are the source of all the great content that drives the network. These guys (and gals) need each other, and they both deserve special treatment. **We design and optimize for two classes of users: anonymous, and logged in.** Consider the following Google Chrome network panel trace on a random Super User question I picked:

	requests	data transferred	DOMContentLoaded	onload
Logged in (as me)	29	233.31 KB	1.17 s	1.31 s
Anonymous	22	111.40 KB	768 ms	1.28 s

We minimize the footprint of HTML, CSS and Javascript for anonymous users so they get their pages *even faster*. We load a stub of very basic functionality and dynamically “rez in” things like editing when the user focuses the answer input area. For logged in users, the footprint is necessarily larger, but we can also add features for our most avid community members at will without fear of harming the experience of the vast, silent majority of anonymous users.

3. Make Performance a Point of (Public) Pride

Now that we’ve exhausted the Yahoo performance guidance, and made sure we’re serving the absolute minimum necessary to our anonymous users — where else can we go for performance? Back to our code, of course.

When it comes to website performance, there is no getting around one fundamental law of the universe: **you can never serve a webpage faster than it you can render it on the server.** I know, duh. But I’m telling you, it’s very easy to fall into the trap of not

noticing a few hundred milliseconds here and there over the course of a year or so of development, and then one day you turn around and your pages are taking almost a full freaking second to render on the server. It's a heck of a liability to start *1 full second in the hole* before you've even transmitted your first byte over the wire!

That's why, as a developer, you need to put performance right in front of your face on every single page, all the time. That's exactly what we did with our [MVC Mini Profiler](#), which we are contributing back to the world as open source. The simple act of **putting a render time in the upper right hand corner of every page we serve** forced us to fix all our performance regressions and omissions.

	duration (ms)	from start (ms)	query time (ms)
http://superuser.com:80/questions/231273/what...	8.8	+0	
InitCurrentUser	4.3	+5	2 sql 1.2
Check redirects	56	+11.3	1 sql 1.2
Get sorter	7	+67.5	1 sql 6.3
Get Answers with Owner	3	+77.2	1 sql 2.1
GetLinkedQuestions	2.4	+80.3	1 sql 0.6
SqlFetch	7.2	+82.7	1 sql 4
VotesCastJson	5.3	+90	1 sql 4.2
Preload deletion info	17.6	+98.7	2 sql 15.1
Check SuggestedEdits	2.1	+116.3	1 sql 0.9
mainbar	2.5	+119.2	1 sql 1
Body	3.5	+121.5	! 2 sql 1.2
SpecialStatus	9.8	+125.4	! 5 sql 2.5
Answer loop	115.3	+135.4	! 54 sql 28.6

263 ms

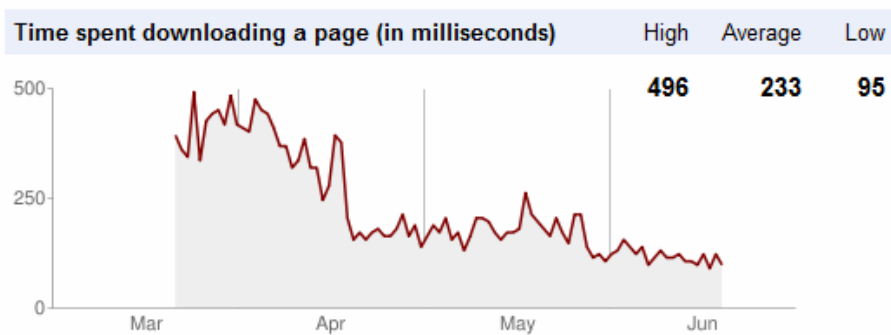
share show trivial show time with children 27.7 % in sql

13 revs 8 users 32% Visit Meta

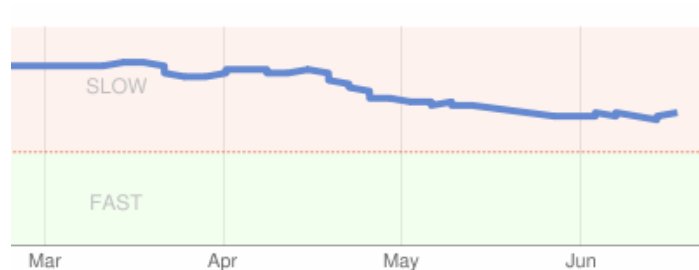
(Note that you can click on the SQL linked above to see what's actually being run and how long it took in each step. *And* you can use the share link to share the profiler data for this run with your fellow developers to shame them diagnose a particular problem. *And* it works for multiple AJAX requests. Have I mentioned that our open source MVC Mini Profiler is totally freaking awesome? If you're on a .NET stack, you should really [check it out](#).)

In fact, with the render time appearing on every page for everyone on the dev team, **performance became a point of pride**. We had so many places where we had just gotten a *little* sloppy or missed some *tiny* thing that slowed a page down inordinately. Most of the performance fixes were trivial, and even the ones that were not turned into fantastic opportunities to rearchitect and make things simpler and faster for all of our users.

Did it work? You bet your sweet [LAsm](#) it worked:



That's the Google crawler page download time; the experimental Google [Site Performance page](#), which ostensibly reflects complete full-page browser load time, confirms the improvements:



While server page render time is only part of the performance story, it is the baseline from which you start. I cannot emphasize enough how much the simple act of putting the page render time on the page helped us, as a development team, build a dramatically faster site. Our site was always relatively fast, but even for a historically “fast” site like ours, we realized huge gains in performance from this one simple change.

I won't lie to you. Performance isn't easy. It's been a long, hard road getting to where we are now - and we've thrown a lot of unicorn dollars toward [really nice hardware](#) to run everything on, though I wouldn't call any of our hardware choices particularly extravagant. And I did [follow my own advice](#), for the record.

I distinctly remember switching from AltaVista to Google back in 2000 in no small part because it was blazing fast. To me, **performance is a feature**, and I simply like using fast websites more than slow websites, so naturally I'm going to build a site that I would want to use. But I think there's also a lesson to be learned here about the competitive landscape of the public internet, where there are two kinds of websites: **the quick and the dead**.

Which one will you be?

Gift this book to your friends...for [free](#).

Become a Hyperink reader. Get a [special surprise](#).

Like the book? Support our author and leave a [comment](#)!

IV.

Hiring Programmers the Right Way



Why Can't Programmers.. Program?

 **Nolan K. Bushnell @NolanBushnell**

"At Atari we hired based on hobbies and not grades in school. We ended up with the best engineering group in the world."

9:18 AM - 3 Aug 11

I was incredulous when I read [this observation from Reginald Braithwaite](#):

Like me, the author is having trouble with the fact that [199 out of 200](#) applicants for every programming job can't write code at all. I repeat: they can't write any code whatsoever.

The author he's referring to is Imran, who is evidently [turning away lots of programmers who can't write a simple program](#):

After a fair bit of trial and error I've discovered that people who struggle to code don't just struggle on big problems, or even smallish problems (i.e. write a implementation of a linked list). They struggle with tiny problems.

So I set out to develop questions that can identify this kind of developer and came up with a class of questions I call "FizzBuzz Questions," named after a game children often play (or are made to play) in schools in the UK. An example of a Fizz-Buzz question is the following:

Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz."

Most good programmers should be able to write out on paper a program which does this in a under a couple of minutes. Want to know something scary? The majority of comp sci graduates can't. I've also seen self-proclaimed senior programmers take more than 10-15 minutes to write a solution.

Dan Kegel [had a similar experience hiring entry-level programmers](#):

A surprisingly large fraction of applicants, even those with master's degrees and PhDs in computer science, fail during interviews when asked to carry out basic programming tasks. For example, I've personally interviewed graduates who can't answer "Write a loop that counts from 1 to 10" or "What's the number after F in hexadecimal?" Less trivially, I've interviewed many candidates who can't use recursion to solve a real problem. These are basic skills; anyone who lacks them probably hasn't done much programming.

Speaking on behalf of software engineers who have to interview prospective new hires, I can safely say that we're tired of talking to candidates who can't program their way out of a paper bag. If you can successfully write a loop that goes from 1 to 10 in every language on your resume, can do simple arithmetic without a calculator, and can use recursion to solve a real problem, you're already ahead of the pack!

Between Reginald, Dan and Imran, I'm starting to get a little worried. I'm more than willing to cut freshly minted software developers slack at the beginning of their career. Everybody has to start somewhere. But **I am disturbed and appalled that any so-called programmer would apply for a job without being able to write the simplest of programs.** That's a slap in the face to anyone who writes software for a living.

The [vast divide between those who can program and those who cannot program](#) is well known. I assumed anyone applying for a job as a programmer had already crossed this chasm. Apparently this is not a reasonable assumption to make. Apparently, FizzBuzz-style screening is *required* to keep interviewers from wasting their time interviewing programmers who can't program.

Lest you think the FizzBuzz test is too easy — and it is blindingly, intentionally easy — a commenter to Imran's post notes its efficacy:

I'd hate interviewers to dismiss [the FizzBuzz] test as being too easy - in my experience it is genuinely astonishing how many candidates are incapable of the simplest programming tasks.

Maybe it's foolish to begin interviewing a programmer without looking at their code first. At Vertigo, we require a code sample before we even proceed to the phone interview stage. And our on-site interview includes a small coding exercise. Nothing difficult, mind you, just a basic exercise to go through the motions of building a small application in an hour or so. Although there have been one or two notable flame-outs, for the most part, this strategy has worked well for us. It lets us focus on actual software

engineering in the interview without [resorting to tedious puzzle questions](#).

It's a shame you have to do so much pre-screening to **have the luxury of interviewing programmers who can actually program**. It'd be funny if it wasn't so damn depressing. I'm [no fan of certification](#), but it does make me wonder if Steve McConnell was onto something with all his talk of [creating a true profession of software engineering](#).

How to Hire a Programmer

There's no magic bullet for hiring programmers. But I can share advice on a few techniques that I've seen work, that I've written about here and personally tried out over the years.

1. First, pass a few simple “Hello World” online tests.

I know it sounds crazy, but some people who call themselves programmers can barely program. To this day, I *still* get regular pings from people who tell me they had candidates [fail the most basic programming test imaginable](#).

That's why *extremely* simple programming tests are [step one of any sane interview process](#). These tests should happen online, and the goal is *not* to prove that the candidate is some kind of coding genius, but that they know what the heck programming is. Yes, it's sad and kind of depressing that this is even necessary, but if you don't perform this sanity check, trust me — you'll be sorry.

Some services that do online code screening (I am sure there are more, but these are the ones I know about) are [Interview Zen](#) and [codility](#).

2. Ask to see their portfolio.

Any programmer worth their salt should [have a portfolio of the things they've worked on](#). It doesn't have to be fancy. I'm just looking for a basic breadcrumb trail of your awesomeness that you've left on the Internet to help others. Show me a Stack Overflow profile where I can see what kind of communicator and problem solver you are. Link me to an open-source code repository of your stuff. Got a professional blog? A tumblr? A twitter? Some other word I've never heard of? Excellent, let's have a look. Share applications you've designed, or websites you worked on, and describe what parts were yours.

Just seeing what kind of work people have done, and what sort of online artifacts they've created, is tremendously helpful in getting a sense of what people do and what they're good (or bad) at.

3. Hire for cultural fit.

[Like GitHub](#), I find that cultural fit is often a stronger predictor of success than mad programming chops.

We talk about [philosophy] during the hiring process, which we take very seriously. We want any potential GitHubber to know what they're getting into and ensure it's a good fit. Part of that is having dinner and talking about stuff like the culture, philosophy, mistakes we've made, plans, whatever.

Early on we made a few hires for their skills with little regard to how they'd fit into the culture of the company or if they understood the philosophy. Naturally, those hires didn't work out. So while we care about the skills of a potential employees, whether or not they "get" us is a major part too.

I realize that not every business *has* a community around what they do, but **if you do have a community, you should try like hell to hire from your community whenever possible**. These are the folks who were naturally drawn to what you do, that were pulled into the gravitational well of your company completely of their own accord. The odds of these candidates being a good cultural fit are abnormally high. That's what you want!

Did a few of your users [build an amazing mod for your game](#)? Did they [find an obscure security vulnerability and try to tell you about it](#)? *Hire these people immediately!*

4. Do a detailed, structured phone screen.

Once you've worked through the above, it's time to give the candidate a call. Bear in mind that the phone screen is not for chatting, it's for *screening*. The call should be technical and structured, so both of you can get out immediately if it clearly isn't a fit. [Getting the Interview Phone Screen Right](#) covers the basics, but in summary:

1. A bit of on-the-fly coding. "Find the largest int value in an int array."
2. Some basic design. "Design a representation to model HTML."
3. Scripting and regular expressions. "Give me a list of the text files in this directory that contain phone numbers in a specific format."
4. Data structures. "When would you use a hashtable versus an array?"
5. Bits and bytes. "Why do programmers think asking if Oct 31 and Dec 25 are the same day is funny?"

What you're looking for is not magical perfect answers, necessarily, but some context into how this person solves problems, and whether they know their stuff (plus or minus 10 percent). The goal is to make sure that the candidates that do make it to the next step are not wasting their time or yours. So don't be shy about sticking to your guns and ending the call early if there are too many warning flags.

(Note: See the next section for a more thorough analysis of the phone screen.)

5. Give them an audition project.

So the candidate breezed through the hello world programming tests, has an amazing portfolio, is an excellent cultural fit, and also passed the phone screen with flying colors. Time to get them in for a face-to-face interview, right? *Not so fast there, cowboy!*

I've seen candidates nail all of the above, join the company, and utterly fail to Get Things Done. Have I mentioned that hiring programmers is hard?

If you want to determine beyond the shadow of a doubt if someone's going to be a great hire, **give them an audition project**. I'm not talking about a generic, abstract programming problem, I'm talking about **a real world, honest-to-God unit of work that you need done right now today on your actual product**. Something you would give to a current employee, if they weren't all busy, y'know, *doing other stuff*.

This should be a regular consulting gig with an hourly rate, and a clearly defined project mission statement. Select a small project that can ideally be done in a few days, maybe at most a week or two. Either the candidate can come in to the office, or they can work remotely. I know not every business has these bite-sized units of work that they can slice off for someone outside the company — but trying desperately to make it *inside* the company — to take on. I'd argue that if you can't think of *any* way to make an audition mini-project work for a strong hiring candidate, perhaps you're not structuring the work properly for your existing employees, either.

If the audition project is a success, fantastic — you now have a highly qualified candidate that can probably Get Things Done, *and* you've accomplished something that needed doing. To date, I have never seen a candidate who passes the audition project fail to work out. I weigh performance on the audition project heavily; it's as close as you can get to actually working the job without being hired. And if the audition project doesn't work out, well, consider the cost of this little consulting gig a cheap exit fee compared to an extensive interview process with 4 or 5 other people at your company. Worst case, you can pass off the audition project to the next strong candidate.

(A probationary period of conditional employment can also work, and is conceptually quite similar. You could hire with a 6-8 week review “go or no go” decision everyone agrees to in advance.)

6. Get in a room with us and pitch.

Finally, you should meet candidates face-to-face at some point. It’s inevitable, but the point of the earlier steps is that **you should be 95 percent certain that a candidate would be a great hire before they ever set foot in an interview room.**

I’m far from an expert on in person interviews, but [I don’t like interview puzzle questions](#), to [put it mildly](#).

Instead, I have my own theory about how we should interview programmers: have the candidate give a 15-minute presentation on their area of expertise. I think this is a far better indicator of success than a traditional interview, because you’ll quickly ascertain ...

- Is this person passionate about what they are doing?
- Can they communicate effectively to a small group?
- Do they have a good handle on their area of expertise?
- Would your team enjoy working with this person?

The one thing every programmer should know, per Steve Yegge, is [how to market yourself, your code, and your project](#). I wholeheartedly agree. *Now pitch me!*

7. None of this is guaranteed.

Please take this list at face value. I’ve seen these techniques work, and I’ve occasionally seen them not work. Adapt this advice to your particular situation, keep what you think makes sense, and ignore the rest (although I’d strongly advise you to never, ever skip step #1). Even in the best of circumstances, **hiring human beings is hard**. A job opportunity may not work out for reasons far beyond anyone’s control. People are, as they say, *complicated*.

If you think of work as a relationship, one you’ll spend 40 hours a week (or more) in the rest of your life, it behooves everyone involved to “date smart.” Both the company and the candidate should make a good faith best effort to determine if there’s a match. Your goal shouldn’t be merely to get a job, or hire someone for a job, but to [have fun](#) and [create a love connection](#). Don’t rush into anything unless it feels right on both sides.

(As an aside, if you're looking for ways to *attract* programmers, you can't go wrong with [this excellent advice from Samuel Mullen](#).)

Getting the Interview Phone Screen Right

It is **very expensive to get the phone screen wrong** — a giant waste of time for everyone involved.



The best phone screen article you'll ever find is Steve Yegge's [Five Essential Phone-Screen Questions](#), another gift to us from Steve's stint at Amazon.

Steve starts by noting two critical mistakes that phone screeners should do their best to avoid:

1. **Don't let the candidate drive the interview.** The interviewer should do most of the talking, guiding the conversation along until they're satisfied the candidate knows the answers to the questions (or has given up).
2. **Watch out for one-trick ponies.** Candidates who only know one particular language or programming environment, and protest complete ignorance of everything else, are a giant red-warning flag.

The point of the phone screen is not for the candidate to drone on about what they've

done. The interviewer should push them out of their comfort zone a bit and ask them *related* questions about things they haven't seen or done before. Ideally, you want to know how this person will react when they face something new, such as *your* codebase.

In an effort to make life simpler for phone screeners, I've put together this list of Five Essential Questions that you need to ask during an SDE screen. They won't guarantee that your candidate will be great, but they will help eliminate a huge number of candidates who are slipping through our process today.

- 1) **Coding.** The candidate has to write some simple code, with correct syntax, in C, C++, or Java.
- 2) **OO design.** The candidate has to define basic OO concepts, and come up with classes to model a simple problem.
- 3) **Scripting and regexes.** The candidate has to describe how to find the phone numbers in 50,000 HTML pages.
- 4) **Data structures.** The candidate has to demonstrate basic knowledge of the most common data structures.
- 5) **Bits and bytes.** The candidate has to answer simple questions about bits, bytes, and binary numbers.

Please understand: **what I'm looking for here is a total vacuum in one of these areas.** It's OK if they struggle a little and then figure it out. It's OK if they need some minor hints or prompting. I don't mind if they're rusty or slow. What you're looking for is candidates who are utterly clueless, or horribly confused, about the area in question.

Of course, you'll want to modify this process to reflect the realities at your shop — so I encourage you to [read the entire article](#). But Steve does provide some examples to get you started:

Coding

Write a function to reverse a string.

Write function to compute Nth fibonacci number.

Print out the grade-school multiplication table up to 12×12.

Write a function that sums up integers from a text file, one int per line.

Write function to print the odd numbers from 1 to 99.

Find the largest int value in an int array.

Format an RGB value (three 1-byte numbers) as a 6-digit hexadecimal string.

Good candidates for the coding problem are verifiably simple, with basic loops or recursion and perhaps a little formatted output or file I/O. All we want to know is whether they really do know how to program or not. Steve's article predates it, but I'd be remiss if I didn't mention [Why Can't Programmers.. Program?](#) here. The FizzBuzz problem is quite similar, and it's shocking how often interviewees can't do it. It's a bit hard to comprehend, like a potential truck driver somehow not being able to find the gas pedal or shift gears.

Object-Oriented Programming

Design a deck of cards that can be used for different card game applications.

Model the Animal kingdom as a class system, for use in a Virtual Zoo program.

Create a class design to represent a filesystem.

Design an OO representation to model HTML.

We're not saying anything about the pros and cons of OO design here, nor are we asking for a comprehensive, low-level OO design. These questions are here to determine whether candidates are familiar with the basic principles of OO, and more importantly, whether the candidate can produce a reasonable-sounding OO solution. We're looking for understanding of the basic principles, as described in [the Monopoly Interview](#).

Scripting and Regular Expressions

Last year my team had to remove all the phone numbers from 50,000 Amazon web page templates, since many of the numbers were no longer in service, and we also wanted to route all customer contacts through a single page.

Let's say you're on my team, and we have to identify the pages having probable U.S. phone numbers in them. To simplify the problem slightly, assume we have 50,000 HTML files in a Unix directory tree, under a directory called "/website". We have 2 days to get a list of file paths to the editorial staff. You need to give me a list of the .html files in this directory tree that appear to contain phone numbers in the following two formats: (xxx) xxx-xxxx and xxx-xxx-xxxx.

How would you solve this problem? Keep in mind our team is on a short (2-day) timeline.

This is an interesting one. Steve says 25 to 35 percent of all software development engineer candidates cannot solve this problem at all — even with lots of hints and given the entire interview hour. What we're looking for is a general reluctance to reinvent the wheel, and some familiarity with scripting languages and regular expressions. To me, this question indicates whether a developer will spend days doing programming work that he or she could have neatly avoided with, perhaps, a quick web search and some existing code that's already out there.

Data Structures

What are some really common data structures, e.g. in java.util?

When would you use a linked list vs. a vector?

Can you implement a Map with a tree? What about with a list?

How do you print out the nodes of a tree in level-order (i.e. first level, then 2nd level, then 3rd level, etc.)

What's the worst-case insertion performance of a hashtable? Of a binary tree?

What are some options for implementing a priority queue?

A candidate should be able to demonstrate a basic understanding of the most common data structures. More specifically, the big ones like arrays, vectors, linked lists, hashables, trees and graphs. They should also know the fundamentals of “big-O” algorithmic complexity: constant, logarithmic, linear, polynomial, exponential and factorial. If they can't, that's a huge warning flag.

Bits and Bytes

Tell me how to test whether the high-order bit is set in a byte.

Write a function to count all the bits in an int value; e.g. the function with the signature `int countBits(int x)`

Describe a function that takes an int value, and returns true if the bit pattern of that int value is the same if you reverse it (i.e. it's a palindrome); i.e. `boolean isPalindrome(int x)`

As Steve says, “Computers don't have ten fingers, they have one. So people need to know this stuff.” You shouldn't be treated to an uncomfortable silence after asking a candidate what 2^{16} is; it's a special number. They should know it. Similarly, they should

know the fundamentals of AND, OR, NOT and XOR — and how a bitwise AND differs from a logical AND. You might even ask about signed vs. unsigned, and why bit-shifting operations might be important. They should be able to explain why the old programmer's joke, "why do programmers think Oct 31 and Dec 25 are the same day?" is funny.

Performing a thorough, detailed phone screen is a lot of work. But it's worth it. Every candidate eliminated through the phone screen saves at least 8 man-hours of time that would have been wasted by everyone in a hands-on test. Each time an unqualified candidate makes it to the hands-on test, you should be asking yourself — **how could we have eliminated this candidate in the phone screen?**

The Years of Experience Myth

There's one aspect of the recruiting process that often goes awry, even with a great phone screen in place. Andrew Stuart of the Australian firm [Flat Rate Recruitment](#) presented an excellent anecdote in an email to me that explains it better than I can:

I had a client building an advanced security application. I sent them person after person and they kept knocking them back. The reason was almost always because the person "didn't have enough low level coding experience." The people I sent had done things like design and develop operating systems, advanced memory managers and other highly sophisticated applications. But my client wasn't interested. They required previous hands-on, low-level coding experience in a particular discipline. Eventually I got an application from a very bright software engineer who almost single-handedly wrote a classic computer emulator, but had little or no low level coding experience in the particular discipline they required.

I told the client, "I have a great guy here who has no experience doing low level coding and I think you should hire him." They were extremely skeptical. I pushed hard to get an interview. "Look, this guy is a superb software engineer who doesn't have low level coding experience in the particular discipline you require now, but if you employ him, within 3-6 months you will have a superb software engineer who does have the low level coding experience you're looking for."

They interviewed him and gave him the job. Within a matter of weeks, it was clear he was the smartest programmer in the company. He quickly mastered their low-level coding and his learning went well beyond that of the other coders in the company. Every time I talk to that client he raves on about this employee, who is now the technical backbone of the company. That company no longer focuses its recruitment on candidates that exactly match previous experience with the required technologies. Instead they focus on finding and employing the smartest and most passionate engineers.

This toxic, counterproductive **years of experience myth** has permeated the software industry for as long as I can remember. Imagine how many brilliant software engineers companies are missing out on because they are completely obsessed with finding people

who match — exactly and to the letter — some highly specific laundry list of skills.

Somehow, they've forgotten that **what software developers do best is learn**. Employers should be looking for passionate, driven, flexible self-educators who have a proven ability to code in *whatever* language — and serving them up interesting projects they can engage with.

It's been shown [time](#) and [time again](#) that **there is no correlation between years of experience and skill in programming**. After about six to twelve months working in any particular technology stack, [you either get it or you don't](#). No matter how many years of "experience" another programmer has under their belt, there's about even odds that *they have no idea what they're doing*. This is why working programmers quickly learn to view their peers with [a degree of world-weary skepticism](#). Perhaps it's the only rational response when the disconnect between experience and skill is so pervasive in the field of software engineering.

With that in mind, do you *really* want to work for a company that still doggedly pursues the years of experience myth in their hiring practices? [Unlikely](#).

Which leads me to my point: Requiring X years of experience on platform Y in your job posting is, well, ignorant. As long as applicants have 6 months to a year of experience, consider it a moot point for comparison. Focus on other things instead that'll make much more of a difference. Platform experience is merely a baseline, not a differentiator of real importance.

In turn that means you as an applicant can use requirements like "3-5 years doing this technology" as a gauge of how clued-in the company hiring is. The higher their requirements for years of service in a given technology, the more likely that they're looking for all the wrong things in their applicants, and thus likely that the rest of the team will be stooges picked for the wrong reasons.

I'm not saying experience doesn't matter in software development. It does. But consider the *entire* range of a developer's experience, and realize that [time invested does not automatically equal skill](#). Otherwise, you may be rejecting superb software engineers simply because they lack "(n) years of experience" in your narrow little technological niche — and that's a damn shame.

On Interviewing Programmers

How do you recognize talented software developers in a 30-minute interview? There's a [roundtable article](#) on this topic at Artima Developer with some good ideas from a group of well-known developers:

- Explore an area of expertise
- Have them critique something
- Ask them to solve a problem (but not a puzzle)
- Look at their code
- Find out what books they read
- Ask about a people problem
- Bring them on for a trial basis

Joel Spolsky has an opinion or two on [interviewing developers](#), which he summarizes as **Smart/Gets Things Done:**

1. Introduction
2. Question about recent project
3. An Impossible Question
4. Write some C Functions
5. Are you satisfied with that code?
6. Design Question
7. The Challenge
8. Do you have any questions?

I definitely don't agree with a few of the things Joel asks here — particularly the low-level C functions. That may have been appropriate for the Excel developers Joel was hiring in 1997, but not these days. I'm also not a huge fan of those abstract impossible questions, eg, "how many optometrists are there in Seattle?" but I suppose that's a matter of taste. If you absolutely must, at least ask an impossible question that has some relevance to a problem your very real customers might encounter. I just can't muster any enthusiasm for completely random arbitrary problems in the face of so many *actual* problems.

Joel recently posted an update questioning the commonly held belief that ["we're only hiring the top 0.5%"](#):

It's pretty clear to me that just because you're hiring the top 0.5% of all applicants for a job, doesn't mean you're hiring the top 0.5% of all software developers. You could be hiring from the top 10% or the top 50% or the top 99% and it would still look, to you, like you're rejecting 199 for every 1 that you hire.

By the way, it's because of this phenomenon — the fact that many of the great people are never on the job market — that we are so aggressive about hiring summer interns. This may be the last time these kids ever show up on the open market. In fact we hunt down the smart CS students and individually beg them to apply for an internship with us, because if you wait around to see who sends you a resume, you're already missing out.

I concur. I've worked with a few interns who were amazing developers. It's a bit like playing the slots, but when you hit the jackpot, you win big. If your company isn't taking advantage of intern programs, start immediately.

Chris Sells also has a mini-blog of sorts [entirely dedicated to interview questions and interview articles](#); I highly recommend it. I'm glad to hear that Microsoft doesn't ask those stupid puzzle questions any more. Who are they trying to hire, [Will Shortz](#)?

I have my own theory about the ideal way to interview developers: **have the candidate give a 20-minute presentation to your team on their area of expertise.** I think this is a far better indicator of success than a traditional interview, because you'll quickly ascertain.

- Is this person passionate about what they are doing?
- Can they communicate effectively to a small group?
- Do they have a good handle on their area of expertise?

- Would your team enjoy working with this person?

Jobs may come and go, but it's the people I've worked with that I always remember.

Hardest Interview Puzzle Question Ever

Have you ever been to an interview for a programming job where they **asked you one of those interview puzzle questions?** I have. The one I got was:

How much of your favorite brand of soda is consumed in this state?

And no, the correct answer is not *who cares*, unless the thing you don't care about is getting the job you're interviewing for. I didn't know it at the time, but this is a Fermi Question.

Puzzle questions were all the rage in programming interviews in the '90s and early aughts. This is documented in the book [How Would You Move Mount Fuji?](#) with a specific emphasis on Microsoft's hiring practices.

HOW WOULD YOU MOVE MOUNT FUJI?

Microsoft's Cult of the Puzzle

HOW THE WORLD'S SMARTEST COMPANIES
SELECT THE MOST CREATIVE THINKERS

WILLIAM POUNDSTONE AUTHOR OF BIG SECRETS



It is prudent to [study common interview puzzle questions](#) if you know you'll be interviewing at a company that asks these sorts of questions. And if you think you're ace at programming puzzle questions, then I challenge you to point your massive brain at this, [the hardest interview puzzle question ever](#):

A hundred prisoners are each locked in a room with three pirates, one of whom will walk the plank in the morning. Each prisoner has 10 bottles of wine, one of which has been poisoned; and each pirate has 12 coins, one of which is counterfeit and weighs either more or less than a genuine coin. In the room is a single switch, which the prisoner may either leave as it is, or flip. Before being led into the rooms, the prisoners are all made to wear either a red hat or a blue hat; they can see all the other prisoners' hats, but not their own. Meanwhile, a six-digit prime number of monkeys multiply until their digits reverse, then all have to get across a river using a canoe that can hold at most two monkeys at a time. But half the monkeys always lie and the other half always tell the truth. Given that the N th prisoner knows that one of the monkeys doesn't know that a pirate doesn't know the product of two numbers between 1 and 100 without knowing that the $N+1$ th prisoner has flipped the switch in his room or not after having determined which bottle of wine was poisoned and what color his hat is, what is the solution to this

puzzle?

In other words, [I hate puzzle questions](#).^{*} And yes, I totally failed that interview. Which was disappointing, because it was kind of a cool job.

Not that [my proposal for interviewing programmers](#) was any more popular, though I do think it's much better.

In the previous section, I presented my own theory about the ideal way to interview developers: **have the candidate give a 10-minute watercooler presentation to your team on something they've worked on**. You'd certainly want to complement this type of interview with some actual hands on programming, to make sure the applicant isn't full of crap — although I'm pretty sure that you can't B.S. your way through a technical presentation to a handful of your peers if you truly have no idea what you're talking about. (And if you can, you should be CEO of a startup by now.)

What I'm optimizing for here is the ability to communicate. Most programmers, once they [pass the FizzBuzz level of competency](#), are decent enough. But coding chops aren't enough. To go from good to great, you must be able to communicate effectively: with your teammates, your manager, the users and ultimately the world.

My wife and I just finished a five-day hospital stay for [the birth of our first child](#). During our stay, we were assisted by a parade of different nurses, at least two different nurses every day, sometimes more as we progressed to different areas of the hospital and through daily shift changes. The quality of care at this particular hospital is generally quite high, but we were flummoxed by the disparity in care between the *worst* nurses and the *best* nurses. After a few days, I finally figured out the common characteristic — **the worst nurses were invariably the worst communicators!** The fact that these nurses couldn't effectively *communicate* with us:

- why they needed to do something
- what the options were
- offer advice
- troubleshoot our problems

Meant they ended up feeling like rigid, inflexible proceduralists who didn't care or constantly had to appeal to authority. Of course, this wasn't true. I'm sure they were perfectly competent registered nurses. But in the absence of reasonable communication,

it sure *seemed* that way. To be fair, these nurses were frequently (but not always!) non-native English speakers.

Hiring is difficult under the best of conditions. But an interview process that relies too heavily on puzzle questions is risky. Sure, you may end up with programmers who can solve (or memorize, I guess) the absolute gnarliest puzzle questions you throw at them. But isn't **effectively communicating those solutions to the rest of the team** important, too? For many programmers, *that's* the hardest part of the puzzle.

* Although I expect aficionados of the style should be able to identify all the classic interview puzzle questions represented here.

Gift this book to your friends...for [free](#).

Become a Hyperink reader. Get a [special surprise](#).

Like the book? Support our author and leave a [comment](#)!

V.

Getting Your Team to Work Together



No Matter What They Tell You, It's a People Problem

 **Jeff Atwood@codinghorror**

“don't hate the programmer, hate the code”

11:16 AM - 18 May 12

Bruce Eckel deftly identifies [the root cause of all software development problems](#):

We are in [a young business](#). Primitive, really — we don't know much about what works, and we keep thinking we've found the silver bullet that solves all problems. As a result, we go through these multi-year boom and bust cycles as new ideas come in, take off, exceed their grasp, then run out of steam. But some ideas seem to have staying power. For example, a lot of the ideas in agile methodologies seem to be making some real impacts in productivity and quality. This is because they focus more on the issues of people working together and less on technologies.

A man I've learned much from, Gerald Weinberg, wrote his first couple of books on the technology of programming. Then he switched, and wrote or coauthored 50 more on the process of programming, and he is most famous for saying “no matter what they tell you, it's always a people problem.”

Usually the things that make or break a project are process and people issues. The way that you work on a day-to-day basis. Who your architects are, who your managers are, and who you are working with on the programming team. How you communicate, and most importantly how you solve process and people problems when they come up. The fastest way to get stuck is to think that it's all about the technology and to believe that you can ram your way through the other things. Those other things are the most likely ones to stop you cold.

Bruce misremembers [the actual quote](#); it's “no matter what the problem is, it's always a people problem.” But Bruce's reformulation has a certain ineffable truthiness to it that is certainly in the spirit of [Gerald Weinberg's writing](#).

Let's say I was tasked with determining [whether your software project will fail](#). With the responses to these three questions in hand, I can tell you with almost utter certainty whether your project will fail:

1. How many [lines of code](#) will your team write?
2. What [kind of software](#) are you building?
3. **Do you like your coworkers?**

That last question isn't a joke. I'm not kidding. Do you like the company of your teammates on a personal level? Do you respect your teammates professionally? If you were starting at another company, would you invite your coworkers along? Do you have spirited team discussions or knock-down, drag-out, last man standing filibuster team arguments? Are there any people on your team you'd "vote off the island" if you could?

It may sound trivial to focus on the people you work with over more tangible things like, say, the actual work, or the particular technology you're using to do that work. But it isn't. **The people you choose to work with are the most accurate predictor of job satisfaction I've ever found.** And job satisfaction, based on my work experience to date, correlates perfectly with success. I have *never* seen a happy, healthy, gelled, socially functional software development team fail. It's a shame such teams are so rare.

As Weinberg said, *it's always a people problem*. If you aren't working with people you like, people you respect, people that challenge and inspire you — then why not? What's stopping you?

Leading By Example

It takes discipline for development teams to benefit from [modern software engineering conventions](#). If your team doesn't have the right kind of engineering discipline, the tools and processes you use are almost irrelevant. I advocated as much in [Discipline Makes Strong Developers](#).

But some commenters were understandably apprehensive about the idea of having a [Senior Drill Instructor Gunnery Sergeant Hartman](#) on their team, enforcing engineering discipline.



You little scumbag! I've got your name! I've got your ass! You will not laugh. You will not cry. You will learn by the numbers. I will teach you.

Cajoling and berating your coworkers into compliance isn't an effective motivational technique for software developers, at least not in my experience. **If you want to pull your team up to a higher level of engineering, you need a leader, not an enforcer.** The goal isn't to brainwash everyone you work with, but to negotiate

commonly acceptable standards with your peers.

I thought Dennis Forbes did an outstanding job of summarizing effective leadership strategies in his post [effectively integrating into software development teams](#). He opens with a hypothetical (and if I know Dennis, probably autobiographical) email that describes **the pitfalls of being perceived as an enforcer**:

I was recently brought in to help a software team get a product out the door, with a mandate of helping with some web app code. I've been trying my best to integrate with the team, trying to earn some credibility and respect by making myself useful.

I've been forwarding various [Joel On Software](#) essays to all, recommending that the office stock up on [Code Complete](#), [Peopleware](#) and [The Mythical Man Month](#), and I make an effort to point out everything I believe could be done better. I regularly browse through the source repository to find ways that other members could be working better.

When other developers ask for my help, I try to maximize my input by broadening my assistance to cover the way they're developing, how they could improve their typing form, what naming standard they use, to advocate a better code editing tool, and to give my educated final word regarding the whole stored procedure/dynamic SQL debate.

Despite all of this, I keep facing resistance, and I don't think the team likes me very much. Many of my suggestions aren't adopted, and several people have replied with what I suspect is thinly veiled sarcasm.

What's going wrong?

I'm sure we've all worked with someone like this. Maybe we were even that person ourselves. Even with the best of intentions, and armed with [the top books on the reading list](#), you'll end up like Gunnery Sergeant Hartman ultimately did: gunned down by your own team.

At the end of his post, Dennis provides [a thoughtful summary of how to avoid being shot by your own team](#):

Be humble. Always first presume that you're wrong. While developers do make mistakes, and as a new hire you should certainly assist others in catching and correcting mistakes, you should try to ensure that you're certain of your observation before proudly declaring your find. It is enormously damaging to your credibility when you cry wolf.

Be discreet with constructive criticism. A developer is much more likely to be accept casual suggestions and quiet leading questions than they are if the same is emailed to the

entire group. Widening the audience is more likely to yield defensiveness and retribution. The team is always considering what your motives are, and you will be called on it and exiled if you degrade the work of others for self-promotion.

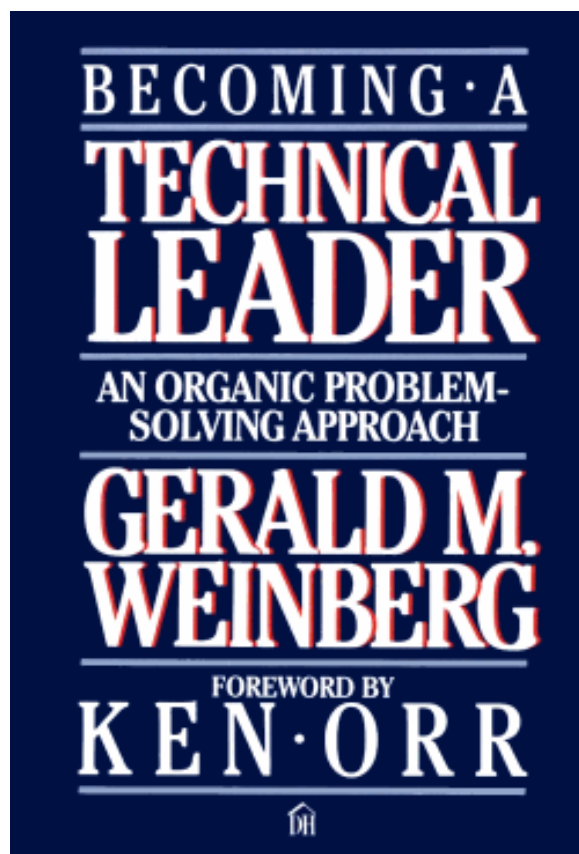
The best way to earn credibility and respect is through hard work and real results. Cheap, superficial substitutes — like best practice emails sent to all, or passing comments about how great it would be to implement some silver bullet — won't yield the same effect, and are more easily neutralized.

Actions speak louder than words. Simply talking about implementing a team blog, or a wiki, or a new source control mechanism, or a new technology, is cheap. Everyone knows that you're just trying to claim ownership of the idea when someone eventually actually does the hard work of doing it, and they'll detest you for it. If you want to propose something, put some elbow grease behind it. For instance, demonstrate the foundations of a team blog, including preliminary usage guidelines, and a demonstration of all of the supporting technologies. This doesn't guarantee that the initiative will fly, and the effort might be for naught, but the team will identify that it's actual motivation and effort behind it, rather than an attempt at some easy points.

There is no one-size-fits-all advice. Not every application is a high-volume e-commerce site. Just because that's the most common best-practices subject doesn't mean that it's even remotely the best design philosophies for the group you're joining.

What I like about Dennis' advice is that it focuses squarely on action and results. It correlates very highly with what I've personally observed to work: **the most effective kind of technical leadership is leading by example**. All too often there are no development leads with the time and authority to enforce, even if they wanted to, so [actions become the only currency](#).

But actions alone may not be enough. You can spend a lifetime learning how to lead and still not get it right. Gerald Weinberg's book [Becoming a Technical Leader: an Organic Problem-Solving Approach](#) provides a much deeper analysis of leadership that's specific to the profession of software engineering.



Within the first few chapters, Weinberg cuts to the very heart of the problem with both Gunnery Sergeant Hartman's and Dennis Forbes' hypothetical motivational techniques:

How do we want to be helped? I don't want to be helped out of pity. I don't want to be helped out of selfishness. These are situations in which the helper really cares nothing about me as a human being. What I would have others do unto me is to love me — not romantic love, of course, but true human caring.

So, if you want to motivate people, either directly or by creating a helping environment, you must first convince them that you care about them, and the only sure way to convince them is by actually caring. People may be fooled about caring, but not for long. That's why the second version of the Golden Rule says, "Love thy neighbor", not "Pretend you love thy neighbor." Don't fool yourself. If you don't really care about the people whom you lead, you'll never succeed as their leader.

Weinberg's [Becoming a Technical Leader](#) is truly a classic. It is, quite simply, the thinking geek's [How to Win Friends and Influence People](#). So much of leadership is learning to give a damn about other people, something that we programmers are notoriously bad at. We may [love our machines and our code](#), but our teammates prove much more complicated.

 **Jeff Atwood@codinghorror:**

“The lesson of Yahoo is that a company with poor leadership is inevitably doomed. “bunch of engineers in a room” is not leadership.”

12:17 PM - 23 May 12

Vampires Programmers versus Werewolves Sysadmins

Kyle Brandt, a system administrator, asks [Should Developers have Access to Production?](#)

A question that comes up [again](#) and [again](#) in web development companies is:

“Should the developers have access to the production environment, and if they do, to what extent?”

My view on this is that as a whole they should have limited access to production. A little disclaimer before I attempt to justify this view is that this standpoint is in no way based on the perceived quality or attitude of the developers — so please don't take it this way.

This is a tricky one for me to answer, because, well, I'm a developer. More specifically, **I'm one of the developers Kyle is referring to**. How do I know that? Because Kyle works for our company, Stack Overflow. And Kyle is a great system administrator. How do I know that? Two reasons:

1. He's one of the top [Server Fault](#) users.
2. He had the audacity to write about this issue on the Server Fault blog.

From my perspective, *the whole point of the company* is to talk about what we're doing. [Getting things done is important](#), of course, but we have to stop occasionally to write up what we're doing, how we're doing it, and why we're even doing it in the first place — including all our doubts and misgivings and concerns. If we don't, we're cheating ourselves, and you guys, out of something much deeper. Yes, writing about what we're doing and explaining it to the community helps us focus. It lets our peers give us feedback. But most importantly of all, it lets *anyone* have the opportunity to learn from our many, many mistakes ... and who knows, perhaps even the occasional success.

That's basically the entire philosophy behind our [Stack Exchange Q&A network](#), too. Let's *all* talk about this stuff in public, so that **we can teach each other how to get better at whatever the heck it is we love to do**.

The saga of System Administrators versus Programmers is not a new one; I don't think I've ever worked at any company where these two factions weren't continually battling with each other in some form. It's truly an epic struggle, but to understand it, you have to appreciate that **both System Administrators and Programmers have different, and perhaps complementary, supernatural powers.**

Programmers are like **vampires**. They're frequently up all night, paler than death itself, and are generally [afraid of being exposed to daylight](#). Oh yes, and they tend think of themselves (or at least their code) as immortal.



System Administrators, however, are like **werewolves**. They may look outwardly ordinary, but are incredibly strong, mostly invulnerable to stuff that would kill regular people — and prone to strange transformations during a moon “outage.”



Let me be very clear that just as Kyle respects programmers, I have [a deep respect for system administrators](#):

Although there is certainly some crossover, we believe that the programming community and the IT/sysadmin community are different beasts. Just because you're a hotshot programmer doesn't mean you have mastered networking and server configuration. And I've met a few sysadmins who could script circles around my code. That's why Server Fault gets its own domain, user profiles, and reputation system.

Different “beasts” indeed.

Anyway, if you're looking for a one size fits all answer to the question of how much access programmers should have to production environments, I'm sorry, I can't give you one. Every company is different, every team is different. I know, it's a sucky answer, but *it depends*.

However, as anyone who has watched [True Blood](#) (or, God help us all, the [Twilight Eclipse](#) movie) can attest, there *are* ways for vampires and werewolves to work together. In a healthy team, everyone feels their abilities are being used and not squandered.

On our team, we're all fair-to-middling sysadmins. But there are a million things to do, and having a professional sysadmin means we can focus on the programming while the networking, hardware and operational stuff gets a whole lot more TLC and far better (read: non-hacky) processes put in place. We're happy to refocus our efforts on what we're expert at, and let Kyle put his skills to work in areas that he's expert at. Now, that

said, we don't want to cede full access to the production servers — but there's a happy middle ground where our access becomes infrequent and minor over time, except in the hopefully rare event of an all-hands-on-deck emergency.

The art of managing vampires and werewolves, I think, is to ensure that they spend their time not fighting amongst themselves, but instead, **using those supernatural powers together to achieve a common goal they could not otherwise**. In my experience, when programmers and system administrators fight, it's because they're bored. You haven't given them a sufficiently daunting task, one that requires the full combined use of their unique skills to achieve.

Remember, it's not vampires versus werewolves. It's vampires *and* werewolves.

Pair Programming versus Code Review

Tom Dommett wrote in to share his positive experience with [pair programming](#):

The idea is two developers work on the same machine. Both have keyboard and mouse. At any given time one is driver and the other navigator. The roles switch either every hour, or whenever really. The driver codes, the navigator is reading, checking, spell-checking and sanity testing the code, whilst thinking through problems and where to go next. If the driver hits a problem, there are two people to find a solution, and one of the two usually has a good idea.

Other advantages include the fact that where two people have differing specialities, these skills are transferred. Ad-hoc training occurs as one person shows the other some tricks, nice workarounds, etcetera.

The end result is that both developers are fully aware of the code, how it works, and why it was done that way. Chances are the code is better than one developer working alone, as there was somebody watching. It's less likely to contain bugs and hacks and things that cause maintenance problems later.

In a bigger team, the pairing can change each week so each team member is partnered with somebody different. This is a huge advantage, as it gets developers talking and communicating ideas in the common language of code.

We found this to be as fast as working separately. The code got written quicker and didn't require revisiting. And when it did need to change, more than one person was familiar with the code.

It's an encouraging result. I applaud anything that gets teams to communicate better.

I'm intrigued by the idea of pair programming, but **I've never personally lived the pair programming lifestyle**. I do, however, enjoy working closely with other developers. Whenever I sit down to work side by side with a fellow developer, I always absorb a few of their tricks and techniques. It's a fast track learning experience for both

participants. But I've only done this in small doses. I'm a little wary of spending a full eight hours working this way. I suspect this might be fatiguing in larger doses, [unless you're very fortunate in your choice of pairing partner](#).

I've [written about the efficacy of code reviews](#) before. That is something I have personal experience with; I can vouch for the value of code reviews without reservation. I can't help **wondering if pair programming is nothing more than code review on steroids**. Not that one is a substitute for the other — you could certainly do both — but I suspect that many of the benefits of pair programming could be realized through [solid peer review practices](#).

But code reviews aren't a panacea, either, [as Marty Fried pointed out](#):

My experience with code reviews has been a mixed bag. One of the problems seems to be that nobody wants to spend the time to really understand new code that does anything non-trivial, so the feedback is usually very general. But later, when someone is working on the code to either add functionality or fix bugs, they usually have lots of feedback (sometimes involving large hammers), but then it may be too late to be effective; the programmer may not even be around. I think it might be useful to have one anyway, but it's hard to get a fellow programmer to tell his boss that another programmer did a bad job.

The advantage of pair programming is its gripping immediacy: it is impossible to ignore the reviewer when he or she is sitting right next to you. Most people will passively opt out if given the choice. With pair programming, that's not possible. Each half of the pair *has* to understand the code, right then and there, as it's being written. Pairing may be invasive, but it can also force a level of communication that you'd otherwise never achieve.

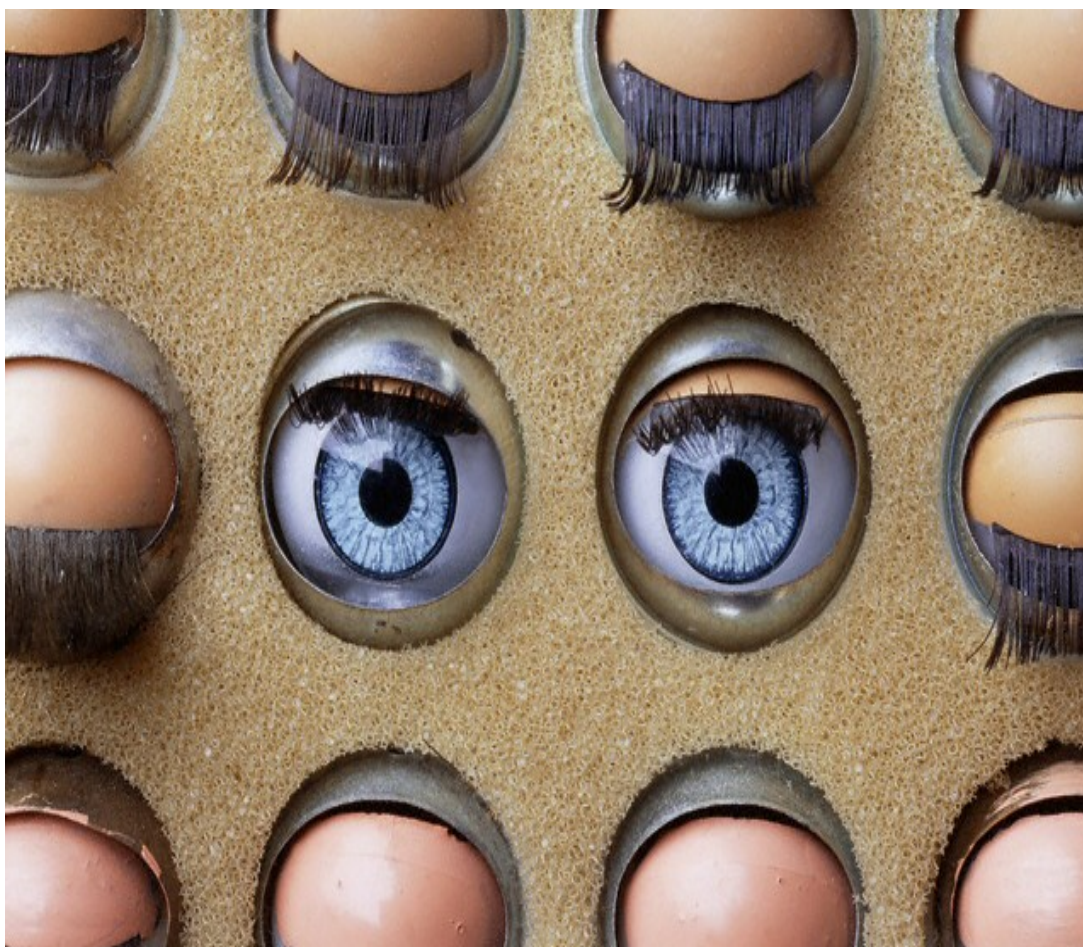
On the other hand, peer review scales a heck of a lot better than stacking physical bodies in the same area. Consider [the experiences of Macadamian with code review](#) while working on the [WINE project](#):

There were two processes in the WINE project that we weren't used to: public peer reviews, where new code and patches were distributed in a mailing list to everyone involved in the project; and single committer, where the project leader had the final say over which patches were accepted into the source tree.

We soon found out that Alexandre Julliard, who has been the maintainer of WINE and one of the key developers since 1994, was very particular about code going into the source tree. Our team's patches were scrutinized, and when some were rejected, there was a lot

of grumbling. “My code works, who does this guy think he is? We’re on a deadline here!” But as the project progressed, we realized we were producing our best code ever. Producing clean, well-designed code that was admitted into the source tree at first pass soon became a matter of pride. We also found that, despite the fact that the project was huge and spread worldwide, we knew exactly how the whole project was progressing since we saw every patch on the mailing list. We now conduct code reviews on every project, and on larger projects, we set up an internal mailing list and designate a single committer. It may be painful to set up code review at your company, and there may be some grumbling, but you will see big improvements in the quality and maintainability of your code.

I think both techniques are clearly a net good, although they each have their particular pros and cons. Is one more effective than the other? Should we do both?



In the end, I don’t think it’s a matter of picking one over the other so much as **ensuring you have more than one pair of eyes looking at the code you’ve written**, however you choose to do it. When your code is reviewed by another human being — whether that person is sitting right next to you, or thousands of miles away — you

will produce better software. That I can guarantee.

Meetings: Where Work Goes to Die

How many meetings did you have today? This week? This month?

Now ask yourself *how many of those meetings were worthwhile*, versus the work that you could have accomplished in that same time.



This might lead one to wonder [why we even have meetings at all](#).

At GitHub we don't have meetings. We don't have set work hours or even work days. We don't keep track of vacation or sick days. We don't have managers or an org chart. We don't have a dress code. We don't have expense account audits or an HR department.

Now, I'm sure Tom was being facetious when he said that GitHub doesn't have meetings, because I sure as heck saw meeting rooms when I recently visited their offices to [give a talk](#). Who knows, maybe they use them to store all the extra forks.

Although some meetings are inevitable, even necessary, the principle he's advocating here is an important one. **Meetings should be viewed skeptically from the outset, as risks to productivity.** We have meetings because we think we need them, but all too often, meetings are where work ends up going to die. I have a handful of principles

that I employ to keep my meetings useful:

1. **No meeting should ever be more than an hour, under penalty of death.**
2. The first and most important constraint on any meeting is the most precious imaginable resource at any company: *time*. If you can't fit your meeting in about an hour, there is something deeply wrong with it, and you should fix that first. Either it involves too many people, the scope of the meeting is too broad, or there's a general lack of focus necessary to keep the meeting on track. I challenge anyone to remember *anything* that happens in a multi-hour meeting. When all else fails, please *keep it short!*
3. **Every meeting should have a clearly defined mission statement.**
4. What's the mission statement of your meeting? Can you define the purpose of your meeting in a single succinct sentence? I hesitate to recommend having an "agenda" and "agenda items" because the word agenda implies a giant, tedious bulleted list of things to cover. Just make sure the purpose of the meeting is clear to everyone; the rest will take care of itself.
5. **Do your homework before the meeting.**
6. Since your meeting has a clearly defined mission statement, everyone attending the meeting knows in advance what they need to talk about and share, and has it ready to go before they walk into the room. *Right?* That's how we can keep the meeting down to an hour. If you haven't done your homework, you shouldn't be in the meeting. If nobody has done their homework, the meeting should be cancelled.
7. **Make it optional.**
8. "Mandatory" meetings are a cop-out. Everyone in the meeting should be there because they want to be there, or they *need* to be there. One sure way to keep yourself accountable for a meeting is to make everyone optional. Imagine holding a meeting that people actually *wanted* to attend, because it was ... useful. Or interesting. Or entertaining. Now make it happen!
9. **Summarize to-dos at the end of the meeting.**
10. If your meeting never happened, what would the consequences be? If the honest answer to that is almost nothing, then perhaps your meeting has no reason to exist. Any truly productive meeting *causes stuff to happen* as a direct result of the decisions

made in that meeting. You, as a responsible meeting participant, are responsible for keeping track of what *you* need to do — and everyone in the room can prove it by summarizing their to-do list for everyone's benefit before they leave the meeting.

It's not that we shouldn't have meetings, but rather, we need to recognize the inherent risks of meetings and strive to make the (hopefully) few meetings we do have productive ones. Let's work fast, minimize BS, and get to the point.

Dealing With Bad Apples

Robert Miesen sent in this story of a project pathology:

I was part of a team writing an web-based job application and screening system (a job kiosk the customer called it) and my team and our customer signed on to implement this job kiosk using Windows, Apache, PHP5, and the ZendFramework — everyone except one of our team members, who I will refer to as “Joe.” Joe kept advocating the use of JavaScript throughout the technology deliberation phase, even though the customer made it quite clear that he expected the vast majority of the job kiosk to be implemented using a server-side technology and all the validation should be done using server-side technology.

The fact that the customer signed off on this, however, did nothing to deter Joe from advocating JavaScript — abrasively. Every time our project hit a bump in the road, Joe would go off on some tirade on how much easier our lives would be if we were only writing this job kiosk in JavaScript. Joe would constantly bicker about how we were all doing this all wrong because we weren't doing it in JavaScript, not even bother to learn the technologies we were actually using, and, whenever fellow teammates would try and gently bring him back into the fold (usually via email), Joe would just flame the poor guy. At the height of Joe's pro-JavaScript bigotry, he would regularly belt off comments like, “Well, if we had only done it in JavaScript,” to such an extent that the team would have been better off if he had just quit (or was reassigned or fired.)

After reading this story, I had to resist the urge to lean forward, hand placed thoughtfully under my chin, brow furrowed, and ask — [have you tried JavaScript?](#)

Robert thought this story was a cautionary tale about technology dependence, but I see something else: **a problem team member, a classic bad apple.**



I'm sure "Joe" had the best of intentions, but at the point where you're actively campaigning against the project, and working against your teammates — **you're a liability to the project.**

The cost of problem personnel on a project is severe, as noted in Chapter 12 of McConnell's [Rapid Development: Taming Wild Software Schedules](#).

If you tolerate even one developer whom the other developers think is a problem, you'll hurt the morale of the good developers. You are implying that not only do you expect your team members to give their all; you expect them to do it when their co-workers are working against them.

In a review of 32 management teams, Larson and LaFasto found that the most consistent and intense complaint from team members was that their team leaders were unwilling to confront and resolve problems associated with poor performance by individual team members. (Larson and LaFasto 1989). They report that, "more than any other single aspect of team leadership, members are disturbed by leaders who are unwilling to deal directly and effectively with self-serving or noncontributing team members." They go on to say that this is a significant management blind spot because managers nearly always think their teams are running more smoothly than their team members do.

How do we identify problem personnel? It's not difficult as you might think. I had a friend of mine once describe someone on his team as — and this is a direct quote — "a cancer." At the point which you, or anyone else on your team, are using words like *cancer* to describe a teammate, you have a serious project pathology. You don't have to be friends with everyone on your team, [although it certainly helps](#), but a level of basic personal and professional respect is mandatory for any team to function normally.

Steve outlines a few warning signs that you're dealing with a bad apple on your team:

1. They cover up their ignorance rather than trying to learn from their teammates. "I

don't know how to explain my design; I just know that it works." or "My code is too complicated to test." (These are both actual quotes.)

2. They have an excessive desire for privacy. "I don't need anyone to review my code."
3. They are territorial. "No one else can fix the bugs in my code. I'm too busy to fix them right now, but I'll get to them next week."
4. They grumble about team decisions and continue to revisit old discussions long after the team has moved on. "I still think we ought to go back and change the design we were talking about last month. The one we picked isn't going to work."
5. Other team members all make wisecracks or complain about the same person regularly. Software developers often won't complain directly, so you have to ask if there's a problem when you hear many wisecracks.
6. They don't pitch in on team activities. On one project I worked on, two days before our first major deadline, a developer asked for the day off. The reason? He wanted to spend the day at a men's clothing sale in a nearby city — a clear sign he hadn't integrated with the team.

Let me be quite clear on this point: if your team leader or manager isn't dealing with the bad apples on your project, **he isn't doing her job.**

You should never be afraid to remove — or even fire — people who do not have the best interests of the team at heart. You can develop skill, but you can't develop a positive attitude. The longer these disruptive personalities stick around on a project, the worse their effects get. They'll slowly spread poison throughout your project, in the form of code, relationships and contacts.

Removing someone from a team is painful; it's not fun for anyone. But **realizing you should have removed someone six months ago** is far more painful.

The Bad Apple: Group Poison

A recent episode of [This American Life](#) interviewed Will Felps, a professor who conducted a sociological experiment demonstrating the [surprisingly powerful effect of bad apples](#).

Groups of four college students were organized into teams and given a task to complete some basic management decisions in 45 minutes. To motivate the teams, they're told that whichever team performs best will be awarded \$100 per person. What they don't know, however, is that in some of the groups, the fourth member of their team isn't a student. He's an actor hired to play a bad apple, one of these personality types:

1. **The Depressive Pessimist** will complain that the task that they're doing isn't enjoyable, and make statements doubting the group's ability to succeed.
2. **The Jerk** will say that other people's ideas are not adequate, but will offer no alternatives himself. He'll say, "you guys need to listen to the expert: me."
3. **The Slacker** will say "whatever," and "I really don't care."

The conventional wisdom in the research on this sort of thing is that none of this should have had much effect on the group at all. Groups are powerful. Group dynamics are powerful. And so groups dominate individuals, not the other way around. There's tons of research, going back decades, demonstrating that people conform to group values and norms.

But Will found the opposite.

Invariably, groups that had the bad apple would perform worse. And this despite the fact that people were in some groups that were very talented, very smart, very likeable. Felps found that the bad apple's behavior had a profound effect — groups with bad apples performed 30 to 40 percent worse than other groups. On teams with the bad apple, people would argue and fight, they didn't share relevant information, they communicated less.

Even worse, **other team members began to take on the bad apple's characteristics.** When the bad apple was a jerk, other team members would begin acting like jerks. When he was a slacker, they began to slack, too. And they wouldn't act

this way just in response to the bad apple. They'd act this way to each other, in sort of a spillover effect.

What they found, in short, is that **the worst team member is the best predictor of how any team performs**. It doesn't seem to matter how great the best member is, or what the average member of the group is like. It all comes down to what your worst team member is like. The teams with the worst person performed the poorest.

The actual [text of the study](#) is available if you're interested. However, I highly recommend [listening to the first 11 minutes of the This American Life show](#). It's a fascinating, highly compelling recap of the study results. I've summarized, but I can't really do it justice without transcribing it all here.

Ira Glass, the host of This American Life, found Felps' results so striking that he began to question his *own* teamwork:

I've really been struck at how common bad apples are. Truthfully, I've been kind of haunted by my conversation with Will Felps. Hearing about his research, you realize just how easy it is to poison any group [...] each of us have had moments this week where we wonder if we, unwittingly, have become the bad apples in our group.

As always, self-awareness is the first step. If you can't tell who the bad apple is in your group, *it might be you*. Consider your own behavior on your own team — are you slipping into any of these negative bad apple behavior patterns, even in a small way?

But there was a solitary glimmer of hope in the study, one particular group that bucked the trend:

There was one group that performed really well, despite the bad apple. There was just one guy, who was a particularly good leader. And what he would do is ask questions, he would engage all the team members, and diffuse conflicts. I found out later that he's actually the son of a diplomat. His father is a diplomat from some South American country. He had this amazing diplomatic ability to diffuse the conflict that normally would emerge when our actor, Nick, would display all this jerk behavior.

This apparently led Will to his next research project: can a group leader change the dynamics and performance of a group by [going around and asking questions](#), soliciting everyone's opinions, and making sure everyone is heard?

While it's depressing to learn that a group can be so powerfully affected by the worst tendencies of a single member, it's heartening to know that a skilled leader, if you're

lucky enough to have one, can intervene and potentially control the situation.

Still, the obvious solution is to address the problem at its source: **get rid of the bad apple.**

Even if it's you.

On Working Remotely

When I first [chose my own adventure](#), I didn't know what working remotely from home was going to be like. I had never done it before. As *programmers* go, I'm fairly social. Which still means I'm a borderline sociopath by normal standards. All the same, I was worried that I'd go stir-crazy with no division between my work life and my home life.

Well, I haven't gone stir-crazy yet. I think. But in building Stack Overflow, I have learned a few things about what it means to work remotely — at least when it comes to programming. Our current team encompasses 5 people, distributed all over the USA, along with the team in NYC.



My first mistake was [attempting to program alone](#). I had weekly calls with my business partner, [Joel Spolsky](#), which were quite productive in terms of figuring out what it was we were trying to do together — but he wasn't writing code. I was coding alone. Really alone. One guy working all by yourself alone. This didn't work *at all* for me. I was unmoored,

directionless, suffering from analysis paralysis, and barely able to get motivated enough to write even a few lines of code. I rapidly realized that I'd made a huge mistake in not [having a coding buddy](#) to work with.

That situation [rectified itself soon enough](#), as I was fortunate enough to find one of my favorite old coding buddies was available. Even though Jarrod was in North Carolina and I was in California, the shared source code was the mutual glue that stuck us together, motivated us, and kept us moving forward. To be fair, we also had the considerable advantage of prior history, because we had worked together at a previous job. But the minimum bar to work remotely is to find **someone who loves code as much as you do**. It's ... enough. Anything else on top of that — old friendships, new friendships, a good working relationship — is icing that makes working together all the sweeter. I eventually expanded the team in the same way by adding another old coding buddy, Geoff, who lives in Oregon. And again by adding Kevin, who I didn't know, but had built amazing stuff for us *without even being asked to*, from Texas. And again by adding Robert, in Florida, who I also didn't know, but spent so much time on every single part of our sites that I felt he had been running alongside our team the whole way.

The reason remote development worked for us, in retrospect, wasn't just shared love of code. I picked developers who I knew — I had incontrovertible *proof* — were amazing programmers. I'm not saying they're perfect, far from it, merely that they were top programmers by any metric you'd care to measure. *That's* why they were able to work remotely. Newbie programmers, or competent programmers who are phoning it in, are absolutely not going to have the moxie necessary to get things done remotely — at least, not without a pointy haired manager, or grumpy old team lead, breathing down their neck. Don't even *think* about working remotely with anyone who doesn't freakin' *bleed* ones and zeros, and has a proven track record of getting things done.

While Joel certainly had a lot of high level input into what Stack Overflow eventually became, I only talked to him once a week, at best (these calls were [the genesis of our weekly podcast series](#)). **I had a strong, clear vision of what I wanted Stack Overflow to be, and how I wanted it to work.** Whenever there was a question about functionality or implementation, my team was able to rally around me and collectively make decisions we liked, and that I personally felt were in tune with this vision. And if you know me at all, you know [I'm not shy about saying no](#), either. We were able to build exactly what we wanted, exactly how we wanted.

Bottom line, we were [on a mission from God](#). And we still are.

So, there are a few basic ground rules for remote development, at least as I've seen it

work:

- The minimum remote team size is two. Always have a buddy, even if your buddy is on another continent halfway across the world.
- Only grizzled veterans who absolutely *love* to code need apply for remote development positions. Mentoring of newbies or casual programmers simply doesn't work at all remotely.
- To be effective, remote teams need full autonomy and a leader (PM, if you will) who has a strong vision *and* the power to fully execute on that vision.

This is all well and good when you have a remote team size of *three*, as we did for the bulk of Stack Overflow development. And all in the same country. [Now we need to grow the company](#), and I'd like to grow it in distributed fashion, by hiring other amazing developers from around the world, many of whom I have met through Stack Overflow itself.

But how do you scale remote development? Joel had some deep seated concerns about this, so I tapped one of my heroes, Miguel de Icaza — who I'm proud to note is on [our all-star board of advisors](#) — and he was generous enough to give us some personal advice based on his experience running the [Mono project](#), which has dozens of developers distributed all over the world.



At the risk of summarizing mercilessly (and perhaps too much), I'll boil down Miguel's advice the best I can. There are three tools you'll need in place if you plan to grow a large-ish and still functional remote team:

1. Real time chat

When your team member lives in Brazil, you can't exactly walk by his desk to ask him a quick question, or bug him about something in his recent checkin. Nope. You need a way

to *casually* ping your fellow remote team members and get a response back quickly. This should be low friction and available to all remote developers at all times. IM, IRC, some web based tool, laser beams, smoke signals, carrier pigeon, two tin cans and a string: whatever. As long as everyone really *uses* it.

We're currently experimenting with [Campfire](#), but whatever floats your boat and you can get your team to consistently *use*, will work. Chat is the most essential and omnipresent form of communication you have when working remotely, so you need to make absolutely sure it's functioning before going any further.

2. Persistent mailing list

Sure, your remote team may know the details of *their* project, but what about all the other work going on? How do they find out about that stuff or even know it exists in the first place? You need a virtual bulletin board: a place for announcements, weekly team reports, and meeting summaries. This is where a classic old-school mailing list comes in handy.

We're using [Google Groups](#) and although it's old school in spades, it works plenty well for this. You can get the emails as they arrive, or view the archived list via the web interface. One word of caution, however. Every time you see something arrive in your inbox from the mailing list you better believe, in your heart of hearts, that it contains useful information. The minute the mailing list becomes just another "whenever I have time to read that stuff", noise engine, or distraction from work ... you've let someone cry wolf too much, and ruined it. So be very careful. Noisy, argumentative, or useless things posted to the mailing list should be punishable by death. Or noogies.

3. Voice and video chat

As much as I love ASCII, sometimes faceless ASCII characters just aren't enough to capture the full intentions and feelings of the human being behind them. When you find yourself sending kilobytes of ASCII back and forth, and still are unsatisfied that you're *communicating*, you should instill a reflexive habit of "going voice" on your team.

Never underestimate the power of actually *talking* to another human being. I know, I know, the whole reason we got into this programming thing was to *avoid* talking to other people, but bear with me here. You can't be face to face on a remote team without flying 6-plus hours, and who the heck has that kind of time? I've got work I need to get done! Well, the next best thing to hopping on a plane is to fire up [Skype](#) and have a little voice chat. Easy peasy. All that human nuance which is totally lost in faceless ASCII characters (yes, even with our old pal [*<:-\)](#)) will come roaring back if you *regularly* schedule voice

chats. I recommend at least once a week at an absolute minimum; they don't have to be long meetings, but it sure helps in understanding the human being behind all those awesome check-ins.

Nobody hates meetings and process claptrap more than I do, but there is a certain amount of process you'll need to keep a bunch of loosely connected remote teams and developers in sync.

1. Monday team status reports

Every Monday, as in [somebody's-got-a-case-of-the](#), each team should produce a brief, summarized rundown of:

- What we did last week
- What we're planning to do this week
- Anything that is blocking us or we are concerned about

This doesn't have to be (and in fact *shouldn't* be) a long report. The briefer the better, but do try to capture all the useful highlights. Mail this to the mailing list every Monday like clockwork. Now, how many "teams" you have is up to you; I don't think this needs to be done at the individual developer level, but you could.

2. Meeting minutes

Any time you conduct what you would consider to be a "meeting" with someone else, take minutes! That is, write down what happened in bullet point form, so those remote team members who couldn't be there can benefit from — or at least hear about — whatever happened.

Again, this doesn't have to be long, and if you find taking meeting minutes onerous then you're probably doing it wrong. A simple bulleted list of sentences should suffice. We don't need to know every little detail, just the big picture stuff: who was there? What topics were discussed? What decisions were made? What are the next steps?

Both of the above should, of course, be mailed out to the mailing list as they are completed so everyone can be notified. You do have a mailing list, right? Of course you do!

If this seems like a lot of jibba-jabba, well, that's because **remote development is hard**. It takes discipline to make it all work, certainly more discipline than piling a bunch

of programmers into the same cubicle farm. But when you imagine what this kind of intellectual work — not just programming, but anything where you're working in mostly thought-stuff — will be like in ten, twenty, even thirty years ... don't you think it will look a lot like what happens every day *right now* on Stack Overflow? That is, a programmer in Brazil helping a programmer in New Jersey solve a problem?

If I have learned anything from Stack Overflow it is that the world of programming is [truly global](#). I am honored to meet these brilliant programmers from every corner of the world, even if only in a small way through a website. Nothing is more exciting for me than the prospect of adding international members to the Stack Overflow team. The development of Stack Overflow should be reflective of what Stack Overflow *is*: an international effort of like-minded — and dare I say *totally awesome* — programmers. I wish I could hire each and every one of you. OK, maybe I'm a little biased. But to me, that's how awesome the Stack Overflow community is.

I believe **remote development represents the future of work**. If we have to spend a little time figuring out how this stuff works, and maybe even make some mistakes along the way, it's worth it. As far as I'm concerned, the future is now. Why wait?

Gift this book to your friends...for [free](#).

Become a Hyperink reader. Get a [special surprise](#).

Like the book? Support our author and leave a [comment](#)!

VI.

Your Batcave: Effective Workspaces for Programmers

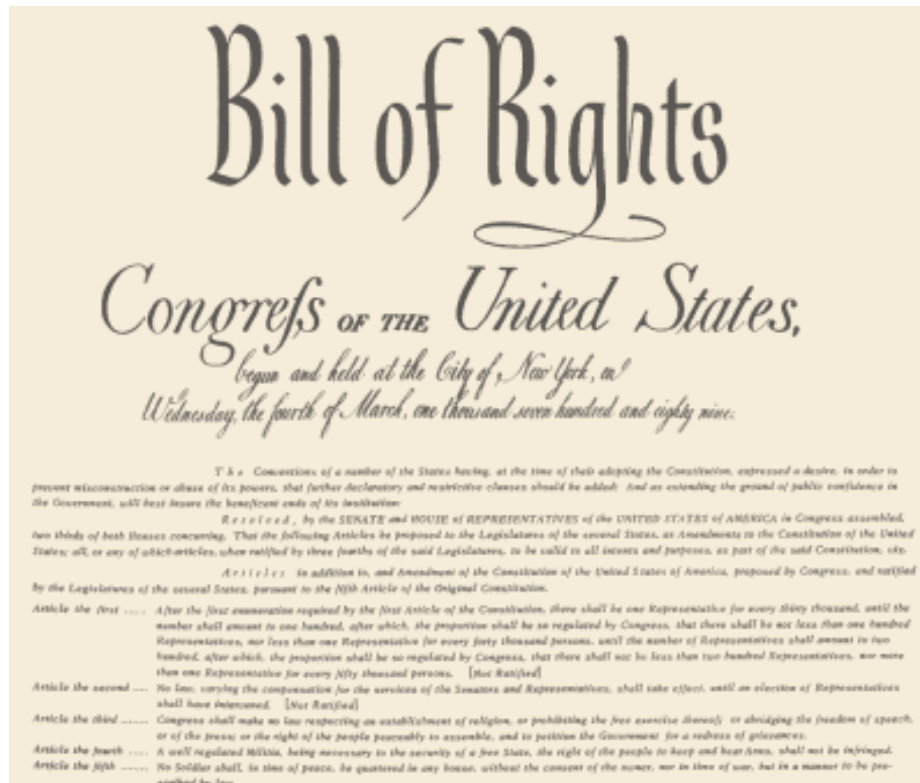


The Programmer's Bill of Rights

“Demand your rights as a programmer! And remember: you can either change your company, or you can *change your company*.”

It's unbelievable to me that a company would pay a developer \$60-\$100k in salary, yet cripple him or her with terrible working conditions and crusty hand-me-down hardware. This makes no business sense whatsoever. And yet I see it all the time. It's shocking how many companies still don't provide software developers with the essential things they need to succeed.

I propose we adopt a **Programmer's Bill of Rights**, protecting the rights of programmers by preventing companies from denying them the fundamentals they need to be successful.



1. Every programmer shall have [two monitors](#)

With the crashing prices of LCDs and the ubiquity of dual-output video cards, you'd be

crazy to limit your developers to a single screen. The productivity benefits of doubling your desktop are [well documented by now](#). If you want to maximize developer productivity, make sure each developer has two monitors.

2. **Every programmer shall have [a fast PC](#)**

Developers are required to run a lot of software to get their jobs done: development environments, database engines, web servers, virtual machines, and so forth. Running all this software requires a fast PC with lots of memory. The faster a developer's PC is, the faster they can cycle through debug and compile cycles. You'd be foolish to pay the extortionist prices for the extreme top of the current performance heap — but always make sure you're buying *near* the top end. Outfit your developers with fast PCs that have lots of memory. Time spent staring at a progress bar is *wasted time*.

3. **Every programmer shall have their choice of [mouse](#) and [keyboard](#)**

In college, I ran a painting business. Every painter I hired had to buy their own brushes. This was one of the first things I learned. Throwing a standard brush at new painters didn't work. The "company" brushes were quickly neglected and degenerated into a state of disrepair. But painters who bought their own brushes took care of them. Painters who bought their own brushes learned to appreciate the difference between the professional \$20 brush they owned and cheap disposable dollar store brushes. Having their own brush engendered a sense of enduring responsibility and craftsmanship. Programmers should have the same relationship with their mouse and keyboard — they are the essential, workaday tools we use to practice our craft and should be treated as such.

4. **Every programmer shall have [a comfortable chair](#)**

Let's face it. We make our livings largely by sitting on our butts for 8 hours a day. Why not spend that 8 hours in a comfortable, well-designed chair? Give developers chairs that make sitting for 8 hours not just tolerable, but enjoyable. Sure, you hire developers primarily for their giant brains, but don't forget your developers' *other* assets.

5. **Every programmer shall have a fast internet connection**

Good programmers [never write what they can steal](#). And the internet is the best conduit for stolen material ever invented. I'm [all for books](#), but it's hard to imagine getting any work done without fast, responsive internet searches at my fingertips.

6. **Every programmer shall have [quiet working conditions](#)**

Programming requires focused mental concentration. Programmers cannot work effectively in an interrupt-driven environment. Make sure your working environment protects your programmers' [flow state](#), otherwise they'll waste most of their time bouncing back and forth between distractions.

The few basic rights we're asking for are easy. They aren't extravagant demands. They're fundamental to the quality of work life for a software developer. If the company you work for isn't getting it right, making it right is neither expensive nor difficult.

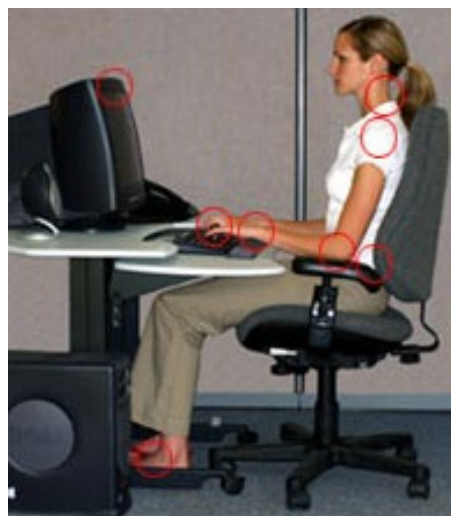
Demand your rights as a programmer! And remember: you can either change your company, or you can *change your company*.

Computer Workstation Ergonomics

I spend almost every waking moment in front of a computer. I'm what you might call an *indoor enthusiast*. **I've been lucky not to experience any kind of computer-related injury due to my prolonged use of computers**, but it is a very real professional risk. I get some occasional soreness in my hands or wrists, mostly after marathon binges where I've clearly overdone it — but that's about the extent of it. All too many of my friends have struggled with long-term [back pain](#) or [hand pain](#). While you can (and should) [exercise your body](#) and [exercise your hands](#) to strengthen them, there's one part of this equation I've been ignoring.

I've been on a quest for the [ultimate computer desk](#) for a few years now, and I've talked at length about the value of [investing in a great chair](#). But I hadn't considered whether my current desk and chair is configured properly to fit my body. What about the **ergonomics** of my computer workstation?

The OSHA has an [official page on computer workstation ergonomics](#), which is a good starting point. But like all government documents, there's a lot more detail here than most people will ever need. The summary picture does give you an idea of what an ergonomic seating position looks like, though. **How close is this to the way you're sitting right now?**



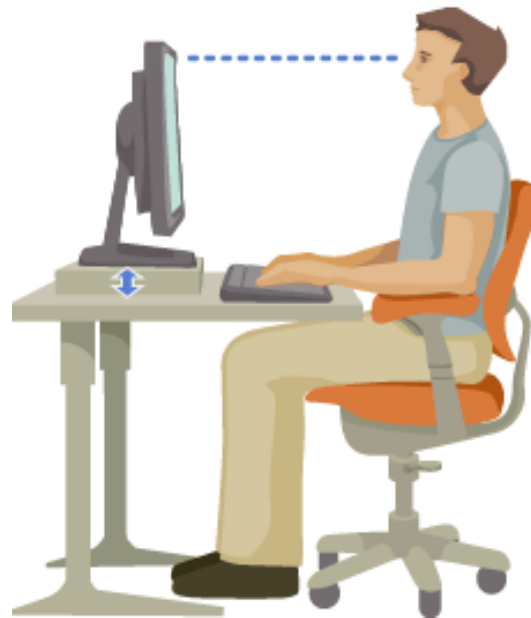
Microsoft doesn't get enough credit for their often innovative hardware division, which [first popularized ergonomic computer input devices](#), starting with the Microsoft Mouse 2.0 in 1993 and following with the Microsoft Natural Keyboard in 1994. With Microsoft's [long-standing interest in hardware ergonomics](#), perhaps it's not too surprising to find that their [healthy computing guide](#) is one of the best and most succinct references for ergonomic computing I've found. But you don't have to read it. I'll summarize the key guidelines for computer workstation ergonomics here, distilling the best advice from *all* the sources I found.

I know I've harped on this, but it bears repeating: **a [quality desk](#) and [quality chair](#) will be some of the best investments you'll ever make as a software developer.** They will last you for 10 years or more, and contribute directly to your work happiness every single day.

If you value your physical health, this is not an area you want to economize on. Hopefully you've invested in a decent computer desk and chair that **provide the required adjustability** to achieve an ergonomically correct computer workstation. Beyond the chair, you'll need to potentially adjust the height of your desk and your monitor, too.



1. The top of your monitor should be at eye level, and directly centered in front of you. It should be about an arm's length in front of you.



2. Your desk surface should be at roughly belly-button level. When your arms are placed on the desk, your elbows should be at a 90-degree angle, just below the desk surface. The armrests of your chair should be at nearly the same level as the desk surface to support your elbows.



3. Your feet should be flat on the floor with your knees at a 90-degree angle. Your seat should not be pressing into the back of your knees; if necessary, tilt it slightly forward to alleviate any knee pressure. Sit fully back in your chair, with your back and shoulders straight and supported by the back of the chair.



4. When typing, your wrists should be in line with your forearms and not bent up, down, or to the side. Your keyboard should be directly centered in front of you. Other frequently used items should be nearby, within arm's reach.



When it comes to computer workstation ergonomics, these are the most basic, most commonly repeated guidelines I've seen. Ergonomics is a holistic discipline, not a science, so your results may vary. Still, I'm surprised how many of these very basic guidelines I've been breaking for so many years, without even thinking about it. I'll be adjusting my home desk tomorrow in hopes of more comfortable computing.

Does More Than One Monitor Improve Productivity?

I've been a multiple monitor enthusiast since the dark days of [Windows Millennium Edition](#). I've written about the manifold joys of many-monitor computing a number of times over the last four years:

- [Multiple Monitors and Productivity](#)
- [Multiple LCDs](#)
- [Joining the Prestigious Three Monitor Club](#)
- [The Large Display Paradox](#)
- [LCD Monitor Arms](#)

I have three monitors at home and at work. I'm what you might call a *true believer*. I'm always looking for **ammunition for fellow developers to claim those second (and maybe even third) monitors that are rightfully theirs** under the [Programmer's Bill of Rights](#).



So I was naturally intrigued when I read about [a new multiple monitor study from the University of Utah](#):

Researchers at the University of Utah tested how quickly people performed tasks like editing a document and copying numbers between spreadsheets while using three different computer configurations:

1. single 18-inch monitor
2. single 24-inch monitor
3. two 20-inch monitors

Here's what they found:

- People using the 24-inch screen completed the tasks 52 percent faster than people who used the 18-inch monitor
- People who used the two 20-inch monitors were 44 percent faster than those with the 18-inch ones.

- Productivity dropped off again when people used a 26-inch screen.

I dug around a bit and found [the actual study results](#) or something very close to it, if you're looking for more detail than the summary I've presented above. This isn't the first time the University of Utah has conducted a multiple-monitor study. It's very similar to the [multiple monitor survey they conducted in 2003](#), also under the auspices of NEC. I agree it's a little sketchy to cite a study from a display vendor that advocates — surprise — buying more and bigger displays. But bear in mind they did find diminishing productivity returns with 26-inch displays. This is something I personally experienced, and I dubbed it the [The Large Display Paradox](#). That finding isn't exactly going to endear them to display vendors.

Patrick Dubroy took [a skeptical look at the multiple monitor productivity claims](#) and found several credible sources of data. I'll combine his finds with mine to provide **a one-stop-shop for research data supporting the idea that, yes, having more display space would in fact make you more productive:**

- [The Virtues of a Second Screen](#)
- [A Comparison of Single and Dual Traditional Aspect Displays with a Widescreen Display over Productivity](#)
- [Toward Characterizing the Productivity Benefits of Very Large Displays](#)
- [The 30-inch Apple Cinema HD Display Productivity Benchmark](#)

Patrick, despite his skepticism — and remember, this is a guy who didn't see a productivity difference between a 14-inch laptop display and a “big ass LCD” — came away convinced:

After looking at the studies, I think it's fair to say that some tasks can be made significantly faster if you have more screen real estate. On the other hand, I think it's clear that most programmers are not going to be 50 percent more productive over the course of a day just by getting a second monitor. The tasks that can be improved are not the bottleneck to programmer productivity.

I'm not sure what Patrick was expecting here. Let me be perfectly clear on this matter: **more is more**. More usable desktop space reduces the amount of time you spend on window management. Instead of incessantly dragging, sizing, minimizing and maximizing windows, you can do actual productive work. With a larger desktop, you can spend less time mindlessly arranging information, and more time interacting with and acting on that

information. How much that matters to you will depend on your job and working style. Personally, I'd be ecstatic if I never had to size, position, or arrange another damn window for the rest of my life.

Choose own your path to happiness, whether it's upgrading to a single 30" display, dual 24" widescreen displays, or three standard 20" displays. As long as it results in more usable desktop space, it's a clear win. **I support all of the above scenarios, and more importantly, the existing research does too.** The price of a few monitors is negligible when measured against the labor cost of a programmer or information worker salary. Even if you achieve a meager two or three percent performance increase, it will have *more* than paid for itself.

What does get a little frustrating is when people claim that one large monitor should be "enough for anyone." This isn't a zero-sum game. Where there is one large monitor, there *could* be two large monitors, or three.

Sometimes, **more is more.**

Coding Horror commenter lomaxx

The biggest advantage I find from having multiple monitors is when I'm referencing data from the second monitor for use on my main monitor. You'd be surprised how often you do this sort of interaction between two windows and having the second monitor is invaluable.

August 13 '08 at 3:42

Investing in a Quality Programming Chair

In [A Developer's Second Most Important Asset](#), I described how buying a quality chair may be one of the **smartest investments you can make as a software developer**.

In fact, after browsing chairs for the last few years of my career, I've come to one conclusion: you can't expect to get a decent chair for less than \$500. If you are spending less than that on seating — unless you are getting the deal of the century on dot-bomb bankruptcy auctions — *you're probably making a mistake*.

I still believe this to be true, and I urge any programmers reading this to seriously consider the value of what you're sitting in while you're on the job. In our profession, seating *matters*:

- **Chairs are a primary part of the programming experience.** Eight hours a day, every day, for the rest of your working life — you're sitting in one. Like it or not, whatever you're sitting in has a measurable impact on your work experience.
- **Cheap chairs suck.** Maybe I've become spoiled, but I have yet to sit in a single good, cheap chair. In my experience, the difference between the really great chairs and the cheap stuff is enormous. A quality chair is so comfortable and accommodating it effortlessly melts into the background, so you can focus on your work. A cheesy, cheap chair constantly reminds you how many hours of work you have left.
- **Chairs last.** As I write this, I'm still sitting my original Aeron chair, which I purchased in 1998. I can't think of any other piece of equipment I use in my job that has lasted me *ten full years* and beyond. While the initial sticker shock of a quality chair may turn you off, try to mentally amortize that cost across the next ten years or more.

Choice of seating is as fundamental and constant as it gets in a programming career otherwise marked by relentless change. They are long-term investments. Why not take the same care and consideration in selecting a chair as you would with the other strategic directions that you'll carry with you for the rest of your career? Skimping yourself on a chair just doesn't make sense.

Although I've been quite happy with my [Herman Miller Aeron chair](#) over the last 10 years, I've always been a little disenchanted with the way it became associated with [dot-com excess](#):

In the '90s, the Aeron became an emblem of the dot-com boom; it symbolized mobility, speed, efficiency, and 24/seven work weeks. The Aeron was a must-have for hot startups precisely because it looked the least like office furniture: It was more like a piece of machinery or unadorned engineering. The black Pellide webbing was durable, and hid whatever Jolt or Red Bull stains you might get on it. Held taut by an aluminum frame, the mesh allowed air to circulate and kept your body cool. What's more, the chair came in three sizes, like a personalized tool. Assorted knobs and levers allowed you to adjust the seat height, tilt tension, tilt range, forward tilt, arm height, arm width, arm angle, lumbar depth, and lumbar height. The Aeron was high-tech but sexy — which was how the dot-commers saw themselves.



But baby-faced CEOs weren't drawn to the Aeron only for the way it looked. The Aeron was a visual expression of the anti-corporate zeitgeist, a non-hierarchical philosophy about the workplace. An office full of Aerons implicitly rejected the Fortune 500, coat-and-tie, brick-and-mortar model in which the boss sinks back in an overpriced, oversized, leather dinosaur while his secretary perches on an Office Max toadstool taking notes.

I recently had the opportunity to sit in a newer [Herman Miller Mirra chair](#) on a trip, and I was surprised how much more comfortable it felt than my classic Aeron.



The Mirra chair was an excellent recliner, too. I've been disappointed by how poorly the Aeron reclines. I actually broke my Aeron's recline pin once and had to replace it myself. So I've retrained myself not to recline, which is awkward, as I'm a natural recliner.

All this made me wonder if I should retire my Aeron and upgrade to something better. I liked the Mirra, but the comments to [my original chair post](#) have a lot of other good seating suggestions, too. Here are pictures and links to **the chairs that were most frequently mentioned as contenders**, in addition to the Mirra and Aeron pictured above:



[Steelcase Think Chair](#)



[Steelcase Leap Chair](#)



[Ergohuman Mesh Chair](#)



[HumanScale Freedom Chair](#)



[HumanScale Liberty Chair](#)

There were also some lesser known recommendations, such as the [Haworth Zody chair](#), [Nightingale CXO chair](#), [BodyBilt ergo chairs](#), [Hag kneeling chair](#), [NeutralPosture ergo](#), the [Chadwick Chair](#) from the original designer of the Aeron, and something called [the swopper](#).

Chair fit is, of course, a subjective thing. If you're investing \$500+ in a chair, you'd understandably want to be sure it's "the one." The thing to do is find a local store that sells all these chairs and try them all out. Well, good luck with that. Don't even bother with your local big-box office supply chain. Your best bet seems to be **back stores**, as they tend to stock many of the more exotic chairs. Apparently they have a clientele of people who are willing to spend for comfort.

Reviews of individual chairs are relatively easy to find, but aren't particularly helpful in isolation. What we need is a multi-chair review roundup. The only notable roundup I know

of is Slate's late 2005 [Sit Happens: The Search for the Best Desk Chair](#). It's not as comprehensive as I would like, but it does have most of the main contenders. Notably, Slate's winner was the [HumanScale Liberty](#).

Some other helpful resources I've found, both in the comments to this post, and elsewhere:

- a [multiple chair roundup at CrunchGear](#)
- a fantastic [research page on chairs](#) someone compiled
- a [multiple-chair roundup at UNC](#)
- another [chair roundup at Consumer Search](#), as well as a [meta-collection of roundups](#).
- video demos of [Leap](#), and the [HumanScale Freedom / Liberty](#)

If this is all a bit too much [furniture porn](#) for your tastes, I understand. As for me, I'm headed off to my local friendly neighborhood back store to figure out which of these chairs will best replace my aging Aeron. By my calculations, the Aeron cost me about \$7 per month over its ten year lifetime; I figure my continued health and comfort while programming are worth at least that much.

Update: Since people have been asking, I ultimately decided the best fit and feel for me, personally, was the [Herman Miller Mirra chair](#). It's a huge upgrade from my ten-year-old Aeron. It feels like three or four revisions better. For example, the front lip of the seat is adjustable, which addresses one of the major concerns I had with my Aeron — as well as the vastly improved reclining I mentioned above. The only unexpected downside is that the plastic back is a little rough on the skin if you sit, er... shirtless. Although I am very pleased with my new shadow Mirra with citron back ([pic](#)), I urge you to do the research and try the chairs yourself before deciding.

Bias Lighting

We computer geeks like it dark. Really dark. Ideally, we'd be in a cave. A cave ... with an internet connection.



The one thing that we can't abide is direct overhead lighting. Every time the overhead light gets turned on in this room, I feel like a [Gremlin](#) shrieking *Bright light! Bright light!* Oh, how it burns!

But **there is a rational basis for preferring a darkened room.** The light setup in a lot of common computing environments [causes glare on the screen](#):

If your room's lit, as most are, by fittings hanging from the ceiling, you'll be wanting to set up your monitor so that you don't see reflections of the lights in it. The flat screens on many modern monitors (like the excellent Samsung I review here) help, because they reflect less of the room behind you. And anti-reflective screen coatings are getting better and better too. But lots of office workers still just can't avoid seeing one or more ceiling fluoros reflected in their screen.

A good anti-reflective coating can reduce most such reflections to annoyance level only. But if you can see lights reflected in your screen, you can probably also directly see lights over the top of your monitor. Direct line of sight, or minimally darkened reflected line of sight, to light sources is going to give you glare problems.

Glare happens when there are small things in your field of vision that are much brighter than the general scene. Such small light sources can't be handled well by your irises; your eyes' pupil size is matched to the overall scene illumination, and so small light sources will appear really bright and draw lines on your retinas. The more of them there are, and the brighter they are, the more work your eyes end up doing and the sooner they'll get tired.

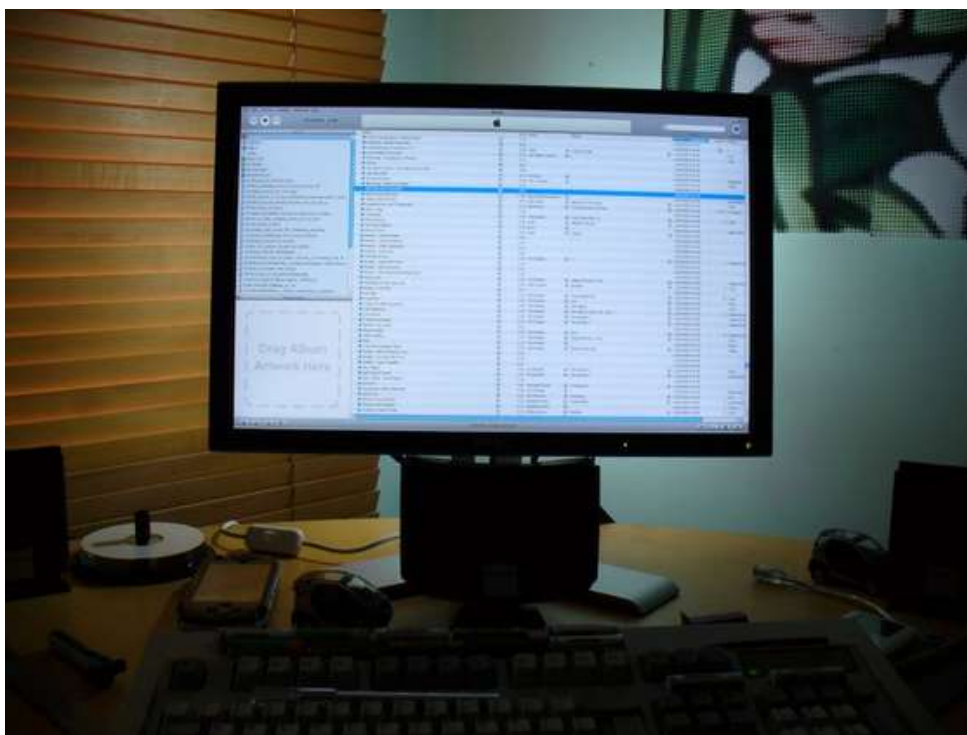
While a darkened room is better for viewing most types of computer displays, it has risks of its own. It turns out that sitting in a dark room staring at a super bright white rectangle is ... kind of bad for your eyes, too. It doesn't help that most LCDs come from the factory [with retina-scorching default brightness levels](#). To give you an idea of how crazy the defaults truly are, the three monitors I'm using right now have brightness set to 25/100. **Ideally, your monitors shouldn't be any brighter than a well-lit book.** Be sure to crank that brightness level down to something reasonable.

You don't want total darkness, what you want is some indirect lighting — specifically bias lighting. It helps your eyes compensate and [adapt to bright displays](#).

"[Bias lighting] works because it provides enough ambient light in the viewing area that your pupils don't have to dilate as far. This makes for less eyestrain when a flashbang gets thrown your way or a bolt of lightning streams across the screen," he told Ars.

"Because the display is no longer the only object emitting light in the room, colors and black levels appear richer than they would in a totally black environment. Bias lighting is key in maintaining a reference quality picture and reducing eye-strain."

Bias lighting is the happy intersection of indirect lighting and light compensation. **It reduces eye strain and produces a better, more comfortable overall computing display experience.**



The good news is that it's trivially easy to set up a bias lighting configuration these days due to the proliferation of inexpensive and bright LEDs. You can build yourself a bias light with [a clamp and a fluorescent bulb](#), or with [some nifty IKEA LED strips](#) and double-sided foam tape.

It really is that simple: **just strap some lights to the back of your monitors.**

I'm partial to the IKEA [Dioder](#) and [Ledberg](#) technique myself; I currently have an array of Ledbergs behind my monitors. But if you don't fancy any minor DIY work, you can try the [Antec Halo 6 LED Bias Lighting Kit](#). It also has the benefit of being completely USB powered.

Of course, lighting conditions are a personal preference, and I'd never pitch bias lighting as a magic bullet. But there is science behind it, it's cheap and easy to try, and **I wish more people who regularly work in front of a computer knew about bias lighting.** If nothing else, I hope this post gets people to turn their LCD monitors down from factory brightness level infinity to something a tad more gentle on the old Mark I Eyeball.

Gift this book to your friends...for [free](#).

Become a Hyperink reader. Get a [special surprise](#).

Like the book? Support our author and leave a [comment!](#)

VII.

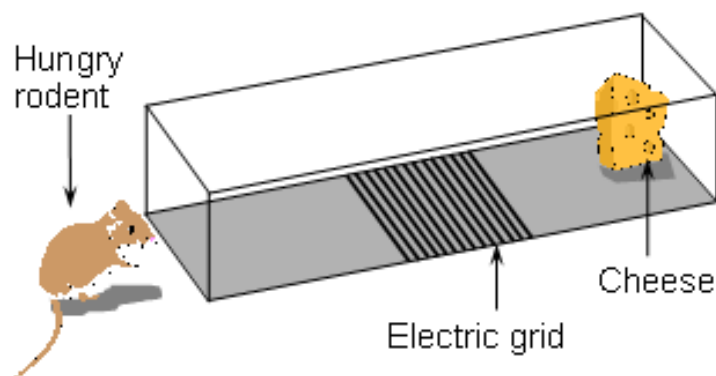
Designing With the User in Mind



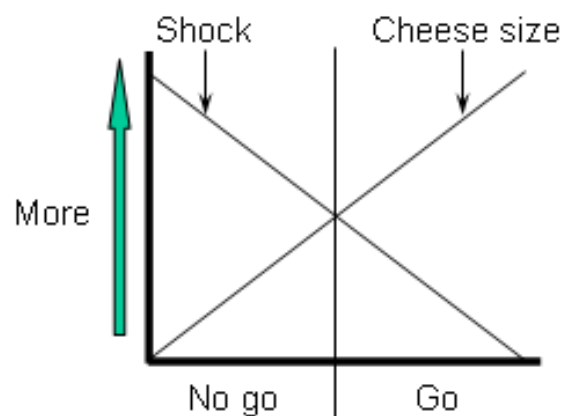
You'll Never Have Enough Cheese

"Getting the details right is the difference between something that delights, and something customers *tolerate*."

This [Human Factors International presentation](#) references something called a Columbia Obstruction Device:



I couldn't find any actual references to the Columbia University science experiment they're referring to, but it certainly seems plausible enough. The parallel with users and usability is natural. Either maximize the cheese (make your application compelling), or minimize the shock (make your application easy to use):



We may think our applications are compelling, but I seriously doubt they look that compelling to users. Unless you are providing users free mp3s, or access to pornography, it's **highly unlikely you will ever have enough cheese to overcome even the**

mildest of electric shocks. The only variable you can really control is your application's usability. The barrier to entry has to be absurdly low to even get people to *look* at your software — much less use it.

This is something that [Joel talks about, too](#):

But there's a scary element of truth to it — scary to UI professionals, at least: an application that does something really great that people really want to do can be pathetically unusable, and it will still be a hit. And an application can be the easiest thing in the world to use, but if it doesn't do anything anybody wants, it will flop. UI consultants are constantly on the defensive, working up improbable ROI formulas about the return on investment clients will get from their \$75,000 usability project, precisely because usability is perceived as "optional," and the scary thing is, in a lot of cases, it is. In a lot of cases. The CNN website has nothing to be gained from a usability consultant.

Napster and **ICQ** were absolute trainwrecks in terms of user interface. But it simply didn't matter. What they delivered was so compelling, and the competition was (at the time) so ineffective, that these developers could get away with terrible UIs.

Delicious cheese is a rare luxury that most developers working on typical business applications will never have. What kind of crazy user looks forward to using a *document management system*? If you want to have any hope at all of users actually using your application, **forget about the cheese**: just make sure you aren't shocking your users.

This is All Your App is: A Collection of Tiny Details

Fair warning: this is a section about automated cat feeders. Sort of. But bear with me, because I'm also trying to make a point about software. If you have a sudden urge to skip on ahead, I don't blame you. I don't often talk about cats, but [when I do, I make it count](#).

We've used automated cat feeders [since 2007](#) with great success. (My apologies for the picture quality, but it was 2007, and camera phones were awful.)



Feeding your pets using robots might sound impersonal and uncaring. Perhaps it is. But I can't emphasize enough **how much of a daily lifestyle improvement it really is to have your pets [stop associating you with ritualized, timed feedings](#)**. As my wife so

aply [explained](#):

I do not miss the days when the cats would come and sit on our heads at 5 AM, wanting their breakfast.

Me neither. I haven't stopped loving our fuzzy buddies, but this was also before we had onetwothree children. We don't have a lot of time for random cat hijinks these days. Anyway, once we set up the automated feeders in 2007, it was a *huge* relief to outsource pet food obsessions to machines. They reliably delivered a timed feeding at 8am and 8pm like clockwork for the last five years. No issues whatsoever, other than changing the three D batteries about once a year, filling the hopper with kibble about once a month, and an occasional cleaning.

Although they *worked*, there were still many details of the automated feeders' design that were downright terrible. I put up with these problems because I was so happy to have automatic feeders that worked at all. So when I noticed that the [2012 version of these feeders](#) appeared to be considerably updated, I went ahead and upgraded immediately on faith alone. After all, it had been nearly five years! Surely the company had improved their product a bit since then ... right? Well, a man can dream, can't he?



When I ordered the new feeders, I assumed they would be a little better than what I had before.



The two feeders don't look so radically different, do they? But pay attention to the details.

- **The food bowl is removable.** It drove me crazy that the food bowl in the old version was permanently attached, and tough to clean as a result.
- **The food bowl has rounded interior edges.** As if cleaning the non-removable bowl of our old version wasn't annoying enough, it also had sharp interior edges, which tended to accrete a bunch of powdered food gunk in there over time. Very difficult to clean properly.
- **The programming buttons are large and easy to press.** In the old version, the buttons were small watch-style soft rubber buttons that protruded from the surface. The tactile feedback was terrible, and they were easy to mis-press because of their size and mushiness.
- **The programming buttons are directly accessible on the face of the device.** For no discernible reason whatsoever, the programming buttons in the old

version were under a little plastic clear protective “sneeze guard” flap, which you had to pinch up and unlock with your thumb before you could do any programming at all. I guess the theory was that a pet could somehow accidentally brush against the buttons and do ... something ... but that seems incredibly unlikely. But most of all, unnecessary.

- **The programming is easier.** We *never* changed the actual feed schedule, but just changing the time for daylight savings was so incredibly awkward and contorted we had to summarize the steps from the manual on a separate piece of paper as a “cheat sheet.” The new version, in contrast, makes changing the time almost as simple as it should be. Almost.
- **There is an outflow cover flap.** By far the number one physical flaw of the old feeder: the feed slot invites curious paws, and makes it [all too easy to fish out kibble on demand](#). You can see in my original photo that we had to mod the feed slot to tape (and eventually bolt) a wire soap dish cover over it so the cats wouldn’t be able to manual feed. The new feeder has a [perfectly aligned outflow flap](#) that I couldn’t even dislodge with my finger. And it works; even our curious-est cat wasn’t able to get past it.
- **The top cover rotates to lock.** On the old feeder, the top cover to the clear kibble storage was a simple friction fit; dislodging it wasn’t difficult, and the cats did manage to do this early on with some experimentation. On the new feeder, the cover is slotted, and rotates to lock against the kibble storage securely. This is the same way the kibble feeder body locks on the base (on both old and new feeders), so it’s logical to use this same “rotate to lock into or out of position” design in both places.
- **The feed hopper is funnel shaped.** The old feed hopper was a simple cylinder, and holds less in the same space as a result. When I transferred the feed over from the old full models (we had literally just filled them the day before) to the updated ones, I was able to add about 15-20 percent more kibble despite the device being roughly the same size in terms of floor space.
- **The base is flared.** Stability is critical; [depending how adventurous your cats are](#), they may physically attack the feeders and try to push them over, or hit them hard enough to trigger a trickle of food dispensing. A flared base isn’t the final solution, but it’s a big step in the right direction. It’s a heck of a lot tougher to knock over a feeder with a bigger “foot” on the ground.
- **It’s off-white.** The old feeder, like the Ford Model T, was available in any color

customers wanted, so long as it was black. Which meant it did a great job of *not* blending in with almost any decor, and also showed off its dust collection like a champ. Thank goodness the new model comes in “linen.”

These are, to be sure, a **bunch of dumb, nitpicky details**. Did the old version feed our cats reliably? Yes, it did. But it was also a pain to clean and maintain, a sort of pain that I endured weekly, for reasons that made no sense to me other than arbitrarily poor design choices. But when I bought the [new version of the automated feeder](#), I was shocked to discover that nearly every single problem I had with the previous generation was addressed. I felt as if the Petmate Corporation™ was actually listening to all the feedback from the people who used their product, and actively refined the product to address *our* complaints and suggestions.

My point, and I do have one, is that details matter. Details matter, in fact, *a hell of a lot*. Whether in automatic cat feeders, or software. As [my friend Wil Shipley once said](#):

This is all your app is: a collection of tiny details.

This is still one of my favorite quotes about software. It’s something we internalized heavily when building Stack Overflow. **Getting the details right is the difference between something that delights, and something customers *tolerate*.**

Your software, your product, is nothing more than a collection of tiny details. If you don’t obsess over all those details, if you think it’s OK to concentrate on the “important” parts and continue to ignore the other umpteen dozen tiny little ways your product annoys the people who use it on a daily basis — you’re not creating great software. Someone else is. I hope for your sake they aren’t your competitor.

The details are hard. Everyone screws up the details at first, just like Petmate did with the first version of this automatic feeder. And it’s OK to screw up the details initially, provided

...

- you’re getting the primary function more or less right.
- you’re listening to feedback from the people who use your product, and actively refining the details of your product based on their feedback every day.

We were maniacal about listening to feedback from avid Stack Overflow users from the earliest days of Stack Overflow in August 2008. Did you know that we didn’t even have *comments* in the first version of Stack Overflow? But it was obvious, based on user feedback and observed usage, that we desperately needed them. There are now, at the

time I am writing this, [1,569 completed feature requests](#); that's more than one per day on average.

Imagine that. Someone who **cares about the details just as much as you do.**

The User Interface is the Application

Shawn Burke's post [Shippin' Ain't Easy](#) (but [somebody gotta do it](#)) explains why you have to resist change at the end of a project, no matter how justifiable and rational the reasons may be. Even the smallest change has a real risk of introducing additional bugs. The first commenter quipped:

TeX doesn't have bugs... Perhaps that's the exception that proves the rule :-)

[Ian Ringrose](#) immediately replied:

But does it have any users? Is the fact that it's very hard to use not a bug in and of itself?

Touche.

Yukihiro Matsumoto, the creator of Ruby, [has strong feelings on this subject](#):

If you have a good interface on your system, and a budget of money and time, you can work on your system. If your system has bugs or is too slow, you can improve it. But if your system has a bad interface, you basically have nothing. It won't matter if it is a work of the highest craftsmanship on the inside. If your system has a bad interface, no one will use it. So the interface or surface of the system, whether to users or other machines, is very important.

It's also something Joel calls [the iceberg secret](#):

I learned this lesson as a consultant, when I did a demo of a major web-based project for a client's executive team. The project was almost 100% code complete. We were still waiting for the graphic designer to choose fonts and colors and draw the cool 3-D tabs. In the meantime, we just used plain fonts and black and white, there was a bunch of ugly wasted space on the screen, basically it didn't look very good at all. But 100% of the functionality was there and was doing some pretty amazing stuff.

What happened during the demo? The clients spent the entire meeting griping about the graphical appearance of the screen. They weren't even talking about the UI. Just the

graphical appearance. "It just doesn't look slick," complained their project manager. That's all they could think about. We couldn't get them to think about the actual functionality. Obviously fixing the graphic design took about one day. It was almost as if they thought they had hired painters.

I had this exact experience on a project recently. We're building all this cool back-end stuff, natch, and we needed a quickie front-end demo app to show it off. So we built a relatively simple demo app. It's decent, but barely competitive with other companies websites.

Guess what the client thought of our project?

I don't care how many kick-ass Visio architecture diagrams you have; as far as the user is concerned, **the UI is the application**. I know [UI is Hard](#), but you *have* to build an impressive UI if you want to be taken seriously. Give your UI the high priority it deserves.

UI-First Software Development

Before I write a single line of code, I want to have a **pretty clear idea of what the user interface will look like first**. I'm in [complete agreement with Rick Schaut here](#):

When you're working on end-user software, and it doesn't matter if you're working on a web app, adding a feature to an existing application, or working on a plug-in for some other application, you need to design the UI first.

This is hard for a couple of reasons. The first is that most programmers, particularly those who've been trained through University-level computer science courses, learned how to program by first writing code that was intended to be run via the command line. As a consequence, we learned how to implement efficient algorithms for common computer science problems, but we never learned how to design a good UI.

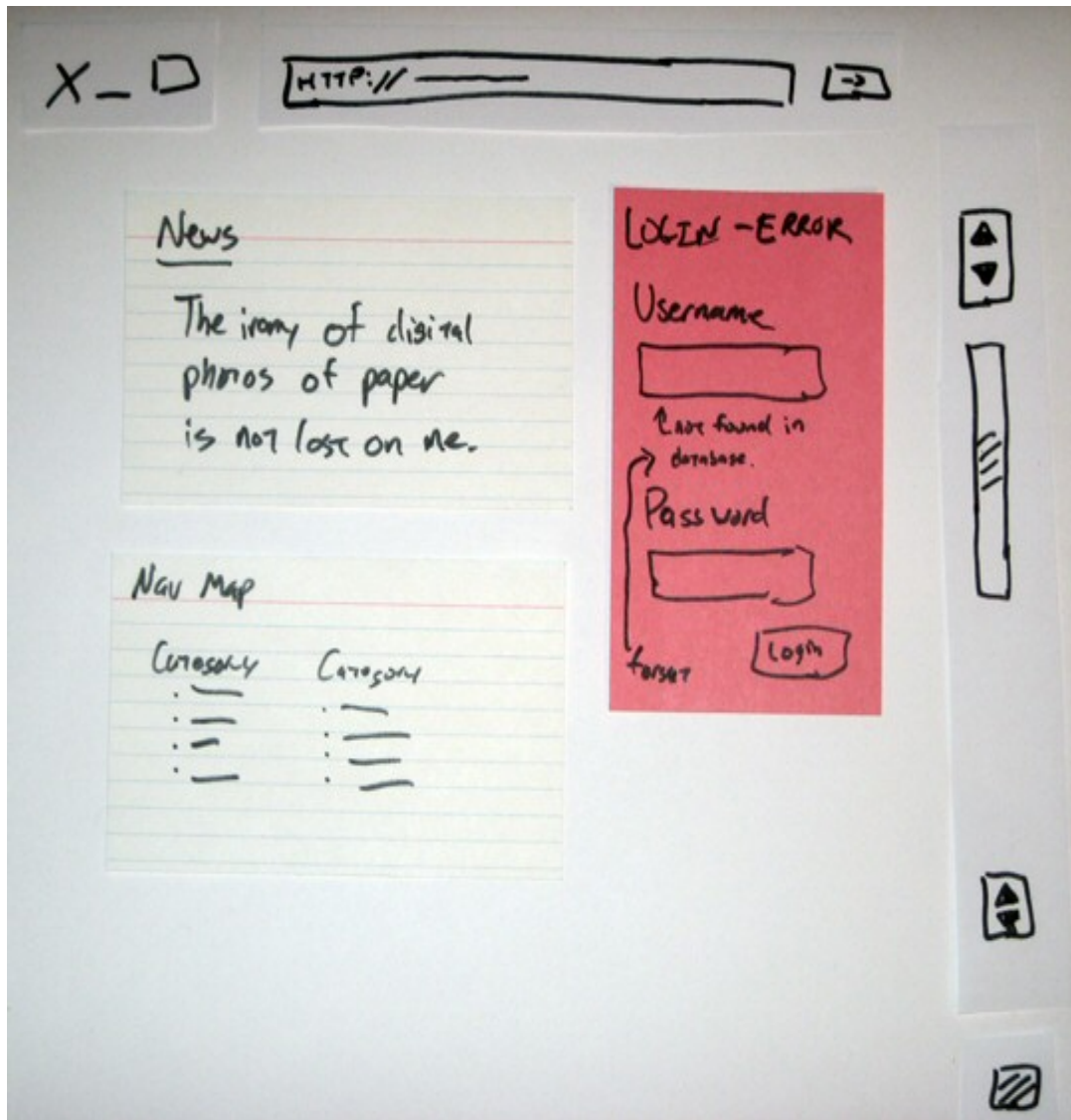
Of course, [UI is hard](#), far harder than coding for developers. It's tempting to skip the tough part and do what comes naturally — start banging away in a code window with no real thought given to how the user will interact with the features you're building.

Remember, to the end user, [the interface is the application](#). Doesn't it make sense to think about that *before* firing up the compiler?

It's certainly true that there are limitations on how the UI can be built based on the technology you're using. Just because some pixels can be arranged a certain way in Photoshop doesn't mean that can magically be turned into a compiling, shippable product in any sane timeframe. To ameliorate that problem, take advantage of [visual design patterns](#). If you're building a GUI application, use a palette of widgets common to your GUI. If you're building a web application, use a palette of HTML, CSS and DOM elements from all over the web. Let the palette enforce your technology constraints.

It shouldn't be difficult to sit down with a few basic tools and slap together a rough mockup of how the user interface will look. However, it is extremely important at this point to **stay out of technical development environments when mocking your user interface**, or the temptation to turn the model into the product may be too strong for your team to resist. Try to avoid [the prototype pitfall](#).

So how do we prototype the UI without relying on our development tools? One way is [simple paper prototyping](#).



The book [Paper Prototyping: The Fast and Easy way to Design and Refine User Interfaces](#) is an excellent introduction to paper prototyping. You can interactively browse sections of this book at [Amazon](#), through [Google Books](#), and [the book's own dedicated web site](#).

There's a certain timelessness to paper prototyping that holds a deep appeal, [as Jacob Nielsen points out](#):

Paper prototyping has a second benefit, besides its impact on your current design project's quality. It will also benefit your career. Consider all the other books you've read about computers, Web design, and similar topics. How much of what you learned will still

be useful in ten years? In twenty years? In the immortal words of my old boss, Scott McNealy, technology has the shelf life of a banana.

In contrast, the paper prototyping technique has a shelf life closer to that of, say, paper. Once you've learned paper prototyping, you can use it in every project you do for the rest of your career. I have no idea what user interface technologies will be popular in twenty years, but I do know that I'll have to subject those designs to usability evaluation, and that paper prototyping will be a valuable technique for running early studies.

Paper prototypes are usually pitched in terms of doing [low-fi usability studies](#), and rightly so. But I find a paper prototype tremendously helpful even if I'm the only one that ever sees it. I need to create an image in my mind of what I'm building, as it will be seen by the world, before I start pouring the concrete to make it real.

If you need any more convincing that paper prototyping is an incredibly valuable tool — even for mere developers — consider the advice of Jared Spool's company, User Interface Engineering:

- [Paper Prototypes: Still Our Favorite](#) (1998)
- [Five Paper Prototyping Tips](#) (2000)
- [Looking Back on 16 Years of Paper Prototyping](#) (2005)

I also recommend reading through [Common Concerns about Paper Prototyping](#) if you're still on the fence.

But what happens when you **outgrow paper prototyping**? Jensen Harris, one of the principal UI designers on the Office 2007 team, first [introduced me to PowerPoint prototyping](#):

We use PowerPoint as kind of a better version of [\[Office 2007\] paper prototypes](#). This technique has several advantages: prototypes can be made to feel somewhat interactive, because the content is electronic it can be modified more easily than paper, and (best of all) the usability participant uses the mouse and is on the computer, so it feels natural to them.

Of course, it doesn't have to be PowerPoint. Use whatever tool you like, as long as it's *not* a development tool. You don't want something too powerful. What you want is mild interactivity while remaining simple and straightforward for quick iterative changes. That's the logical next step up from paper prototyping.



It's a lot easier to share this digital artifact on a distributed team than it is to share a bunch of physical paper. If you're curious about the nuts and bolts of PowerPoint prototyping, dig in:

- [Wireframe prototyping using PowerPoint 2007](#) (Manuel Clement, 26 minute video)
- [Step-by-Step Guide to PowerPoint Prototyping](#) (Jan Verhoeven)
- [PowerPoint Prototyping Toolkit](#) (Long Zheng)

The pursuit of UI-First software development is more important than any particular tool. Use paper, use PowerPoint, [use Keynote](#), use whatever makes sense to you. As long as you avoid, in the words of Manuel Clement, *pouring concrete too early*.

The End of Pagination

What do you do when you have a lot of things to display to the user, far more than can possibly fit on the screen? [Paginate, naturally.](#)



There are plenty of [other real-world examples](#) in this 2007 article, but I wouldn't bother. If you've seen one pagination scheme, you've seen them all. The state of art in pagination hasn't exactly changed much — or at all, really — in the last 5 years.

I can understand paginating when you have 10, 50, 100, maybe even a few hundred items. But once you have *thousands* of items to paginate, **who the heck is visiting page 964 of 3810?** What's the point of paginating so much information when there's a hard practical limit on how many items a human being can view and process in any reasonable amount of time?

Once you have thousands of items, you don't have a pagination problem. You have a search and filtering problem. Why are we presenting hundreds or thousands of items to the user? What does that achieve? **In a perfect world, every search would result in a page with a single item: exactly the thing you were looking for.**

Q

About 858,000,000 results (0.33 seconds)

[I Still Haven't Found What I'm Looking For - Wikipedia, the free ...](#)

en.wikipedia.org/.../I_Still_Haven't_Found_What_I'm_Looking_For

"I Still Haven't Found What I'm Looking For" is a song by rock band U2. It is the second track from their 1987 album The Joshua Tree and was released as the ...

↳ [Writing and recording](#) - [Release](#) - [Live performances](#) - [Reception](#)

But perhaps you *don't* know exactly what you're looking for: maybe you want a variety of viewpoints and resources, or to compare a number of similar items. Fair enough. I have a difficult time imagining any scenario where presenting a hundred or so items wouldn't meet that goal. Even so, the items would naturally be presented in some logical *order* so the most suitable items are near the top.

Once we've chosen a suitable order and a subset of relevant items ... **do we really need pagination at all?** What if we did some kind of endless pagination scheme, where we loaded more items into the view dynamically as the user reaches the bottom? Like so:

It isn't just oddball [disemvowelled](#) companies, either. [Twitter's timeline](#) and [Google's image search](#) use a similar **endless pagination** approach. Either the page loads more items automatically when you scroll down to the bottom, or there's an explicit "show more results" button.

Pagination is also friction. Ever been on a forum where you wished like hell the other people responding to the thread had read all four pages of it before typing their response? Well, maybe some of them would have if the next page buttons weren't so impossibly small, or better yet, *not there at all* because pagination was automatic and seamless. We should be [actively removing friction where we want users to do more of something](#).

I'm not necessarily proposing that all traditional pagination be replaced with endless pagination. But we, as software developers, should **avoid mindlessly generating a list of thousands upon thousands of possible items and paginating it as a lazy one-size-fits-all solution**. This puts all the burden on the user to make sense of the items.

Remember, we invented computers to make the user's life easier, not more difficult.

Once you've done that, there's a balance to be struck, [as Google's research](#) tells us:

User testing has taught us that searchers much prefer the view-all, single-page version of content over a component page containing only a portion of the same information with arbitrary page breaks.

Interestingly, the cases when users didn't prefer the view-all page were correlated with high latency (e.g., when the view-all page took a while to load, say, because it contained many images). This makes sense because we know users are less satisfied with slow results. So while a view-all page is commonly desired, as a webmaster it's important to balance this preference with the page's load time and overall user experience.

Traditional pagination is not particularly user friendly, but endless pagination isn't without its own faults and pitfalls, either:

- The scroll bar, the user's moral compass of "how much more is there?" doesn't work in endless pagination because it is effectively infinite. You'll need an alternate method of providing that crucial feedback, perhaps as a simple percent loaded text docked at the bottom of the page.
- Endless pagination should not break deep linking. Even without the concept of a "page," users should be able to clearly and obviously link to any specific item in the list.
- Clicking the browser forward or back button should preserve the user's position in the endless scrolling stream, perhaps using [pushState](#).
- Pagination may be a bad user experience, but it's *essential* for web spiders. Don't neglect to accommodate web search engines with a traditional paging scheme, too, or perhaps [a Sitemap](#).
- Provide visible feedback when you're dynamically loading new items in the list, so the user can tell that new items are coming, and their browser isn't hung — and that they haven't reached the bottom yet.
- Remember that the user won't be able to reach the footer (or the header) any more, because items keep appearing as they scroll down in the river of endless content. So either move to static headers and footers, or perhaps use the explicit "load more" button instead of loading new content automatically.

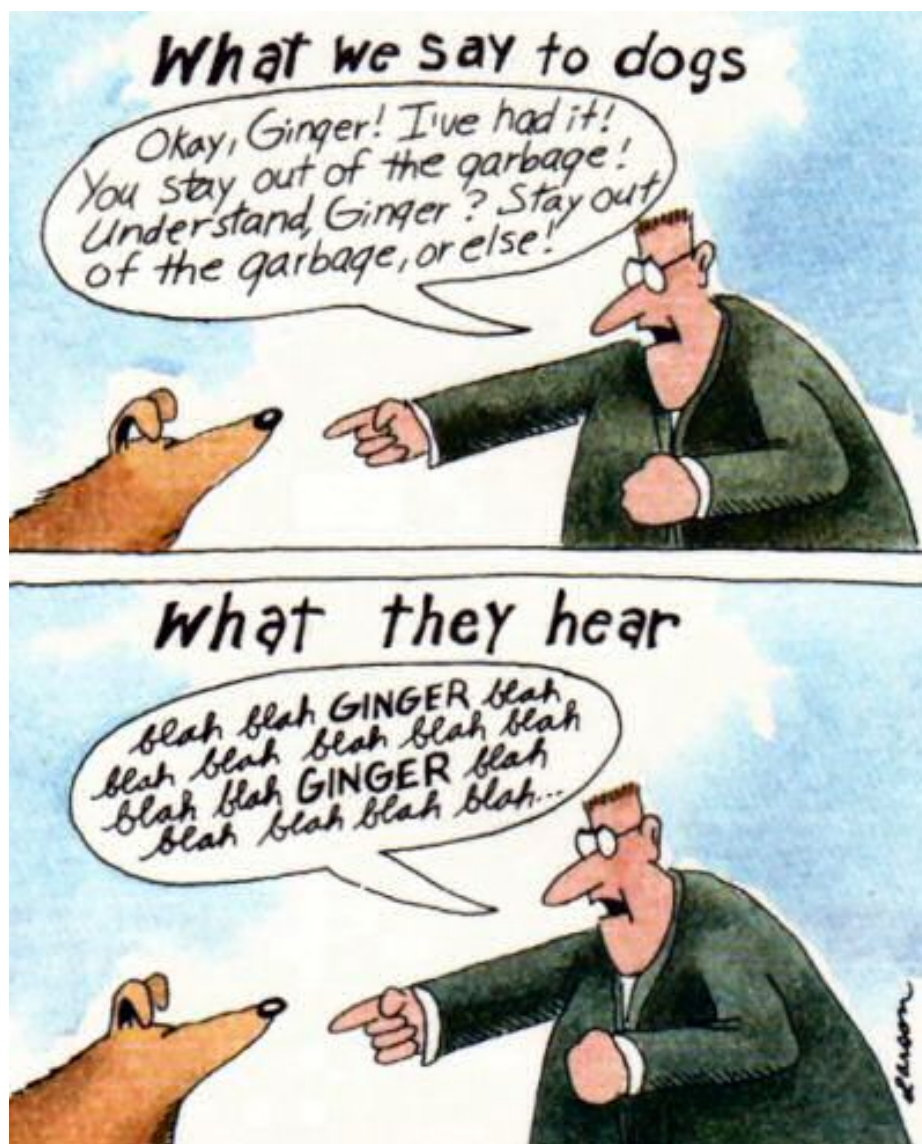
For further reading, there's some excellent [Q&A on the topic of pagination](#) at [ux.stackexchange](#).

Above all else, you should strive to make pagination irrelevant because the user never has to look at more than a few items to find what they need. That's why I suspect Google hasn't done much with this technique in their core search result pages; if they aren't providing great results on page 1, it doesn't really *matter* what kind of pagination they use because they're not going to be in business much longer. Take that lesson to heart: you should be worried most of all about presenting a *relevant* list of items to the user in a sensible order. Once you've got that licked, then and only then should you think about your pagination scheme.

Treating User Myopia

One entry in particular that I keep touching on is [Teaching Users to Read](#). That was specific to dialog boxes, which not only [stop the proceedings with idiocy](#), but are their own delightful brand of user interface poison. Fortunately, you don't see dialogs in web apps much, but this sort of modal dialog lunacy is, sadly, becoming more popular in today's AJAX-y world of web 2.5. Those who can't learn from history are [doomed to repeat it](#), I guess.

Having five more years of development experience under my belt, I no longer believe that classic Larson strip is specific to dialog boxes.



The plain fact is **users will not read anything you put on the screen.**

What we're doing with the trilogy is not exactly rocket surgery. At its core, we run Q&A websites. And the most basic operation of any Q&A website is asking a question. Something any two-year-old child knows how to do.

When we launched superuser.com, that was our fourth Q&A website. This one is for power users, and it's the broadest to date, topic-wise: anything dealing with computer software or hardware (that isn't gaming) is allowed.

We've been at this for over a year now, doing nothing but relentlessly polishing and improving our Q&A engine based on community feedback. We're not particularly good, but we do try very, very hard not to suck. I thought surely, *surely* we must have something as simple as **the ask question form** down by now.

How foolish I was.

Let's take a look at one recent superuser question. I'm presenting it here as it would have been seen by the user who asked the question, while they were entering it on the ask question form.

Title

QuestionVista SP1 32 bit - a network share contains volume mount points that someti

B I [emojis] [icons] [help ?]

I have a Vista SP1 32 bit machine. The machine has a network share that contains volume mount points to the volumes of the vista machine (created using the SetVolumeMountPoint API). When browsing the network share from another computer (either Win7 64 bit, Win7 32 bit or Vista SP1 32 bit) using Windows explorer the following problem occurs:

1. First, both volume mount points called "c", "d" appear fine.
2. I browse into directory "c" and see all its contents properly.
3. I go back to the root of the shared folder and now I only see "d". "c" has disappeared from the directory listing
4. I enter "d", see all its contents, go back to the root of the shared folder and now it's empty. "d" disappeared as well.
5. If I manually go to \\(path to shared folder)\c from the address bar - then all is fine and I can browse its contents. (same with "d")

The same issue does NOT occur when creating a similar share with volume mount points on Windows XP SP2 or SP3.

I have a Vista SP1 32 bit machine. The machine has a network share that contains volume mount points to the volumes of the vista machine (created using the SetVolumeMountPoint API). When browsing the network share from another computer (either Win7 64 bit, Win7 32 bit or Vista SP1 32 bit) using Windows explorer the following problem occurs: 1. First, both volume mount points called "c", "d" appear fine. 2. I browse into directory "c" and see all its contents properly. 3. I go back to the root of the shared folder and now I only see "d". "c" has disappeared from the directory listing 4. I enter "d", see all its contents, go back to the root of the shared folder and now it's empty. "d" disappeared as well. 5. If I manually go to \\c from the address bar - then all is fine and I can browse its contents. (same with "d")

The same issue does NOT occur when creating a similar share with volume mount points on Windows XP SP2 or SP3.

Has anyone came across this problem? Any ideas how to work around it?

Thanks in advance, Barakando

How to Ask

Is your question about computer software or computer hardware?

We prefer questions that can be answered, not just discussed.

Provide details. Write clearly and simply.

If your question is about this website, **ask it on meta** instead.

Formatting Reference

- indent code by 4 spaces
- don't want colorization? Use `<pre>`
- to linebreak use 2 spaces at end
- `>` blockquote
- backtick escapes ``like _this_``
- `<http://foo.com>`
- `[foo](http://foo.com)`
- `foo`

basic HTML also allowed

full reference »

Immediately, there's a problem. **The question formatting is completely wrong!** It's one big jumble of text.

Our formatting rules aren't complicated. You can get a lot done with a bunch of simple paragraphs. [We use Markdown](#), which offers basic formatting conventions that ape ASCII conventions. On top of that, we offer a **real-time preview** of how your question will look once submitted, directly under the question entry area. But none of that seemed to work for this particular asker, who, apparently, was totally satisfied with obviously broken formatting — even though a few choice carriage returns would have worked wonders, and been immediately visible in the live preview.

Yes, yes, it inevitably gets whipped into shape through the collective efforts of our legions of community editors — but that's not the point. It's best if the original asker gets the

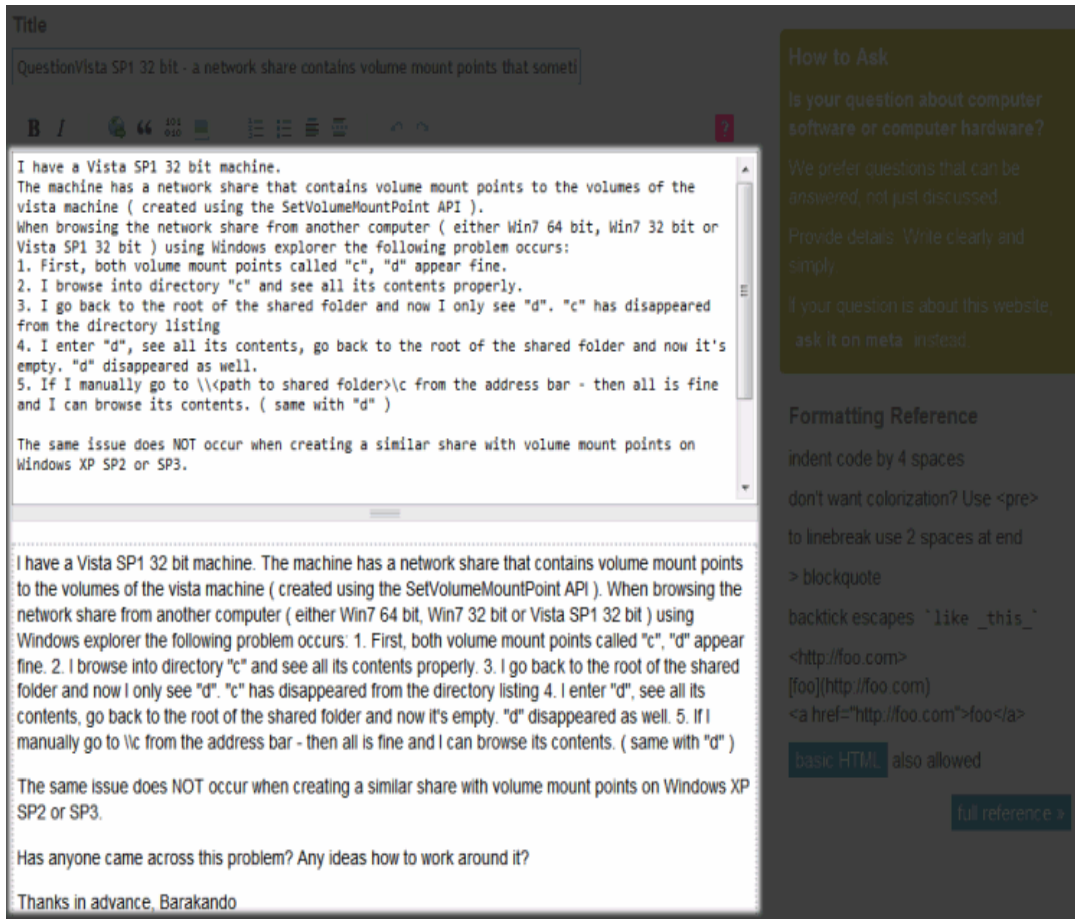
question formatted right to start with, and it is our job as UI designers to make that outcome as statistically likely as we can.

To that end, we've put a bunch of helpful tools on the ask question page to help users get the formatting right. **As UI designers, here's how we see the ask question page:**



We've provided a toolbar with a *neon pink* help button above the question body, and to the right of the question body, we've provided a handy formatting quick reference with a link to the full formatting reference (which opens in a tab / new window by default).

But none of that matters, because **here's how the user sees the ask question page:**



Or rather, here’s everything the user *doesn’t* see.

When I said users don’t read anything you put on the screen, I was lying. Users do read. But **users will only read the absolute minimum amount of text on the screen necessary to complete their task**. I can’t quite explain it, but this kind of user myopia is epidemic. It’s the same problem, everywhere I turn.

How do we treat user myopia? How do we reach these users? The ask question page is already dangerously close to cluttered with helpful tips, but apparently these helpful buttons, links and text are all but invisible to a large segment of the user population. Sure, you could argue that [Super User](#) tends to attract less sophisticated users, but I see the exact same problem with programmers on [Stack Overflow](#). As new users, a significant percentage of them can’t figure out how to format code, even though there’s not only a toolbar button that does it for you, but help text on the right explicitly describing how to do it manually. (Just indent 4 spaces. Spoiler alert!)

More and more, I’m thinking we need to put the formatting help — for new users only — **directly in their line of sight**. That is, pre-populate the question entry area with some example formatting that is typical of the average question. Nothing complicated. But at

least then it'd be in the one — and apparently the *only* one — place myopic users are willing to look. Right in front of their freakin' faces.

The next time you're designing a UI, consider user myopia. You might be surprised just how myopic your users can be. Think long and hard about placing things *directly* in front of them, where they are not just visible, but *unavoidable*. Otherwise they might not be seen at all.

Revisiting The Fold

After I posted my blog entry on [Treating User Myopia](#) I got a lot of advice. Some useful, some not so useful. But the one bit of advice I hadn't anticipated was that we were not making good use of the area "above the fold." This surprised me. **Does the fold still matter?**

The fold refers to the border at the bottom of the browser window at the user's default screen resolution. Like so:



Way back in the dark ages of 1996, it was commonly thought that [users didn't know how to scroll a web page](#).

On the Web, the inverted pyramid becomes even more important since we know from several user studies that users don't scroll, so they will very frequently be left to read only

the top part of an article.

Thus, it was critically important to **cram in as much content in as possible above that fold**, as anything below it was invisible to a huge number of users. They didn't know how to scroll, so they would never find it. Jacob Nielsen, renowned usability expert, is the author of the above quote. But he recanted his position in 2003:

In 1996, I said that “users don't scroll.” This was true at the time: many, if not most, users only looked at the visible part of the page and rarely scrolled below the fold. The evolution of the Web has changed this conclusion. As users got more experience with scrolling pages, many of them started scrolling.

Scrolling is an example [of usability versus learnability](#). It was always my belief that users quickly learned to scroll, otherwise they were permanently crippled as web citizens. If you can't learn to scroll within an hour or so of using the web, you're going to have an awfully stunted experience — so much so that you're probably better off not using it at all. In short, if you use the web, *you know how to scroll*, almost by definition. It is a fundamental skill.

Even today, people will **cite the ancient, irrelevant rule of The Fold** as if it's still law. In fact, I was just talking to a friend of mine who expressed his frustration at dealing with a middle manager who was using the “content must be above the fold” rule as a weapon, and demanding that all page content appear above the fold. It's terribly misguided.

Although thoroughly debunked, there are **still some hidden dangers from the fold**, and subtlety to how users react to it. As documented by [a recent usability study on the fold](#), there are three specific pitfalls to watch out for:

1. **Don't cram everything in above the fold.** Users will explore and find your content — as long as the page “looks” scrollable.
2. **Watch out for stark, horizontal lines that happen to line up with the fold.** This is the only factor that causes users to stop scrolling, because the page looks done and complete. Instead, have a small amount of content just visible, poking up above the fold. This encourages scrolling.
3. **Avoid in-page scroll bars.** The standard browser scrollbar is an indicator of the amount of content on the page that users learn to rely on. Placing `<iframe>` and other elements with scroll bars on the page can break this convention — and may lead to users not scrolling.

These are excellent guidelines, [backed by actual eye tracking and experimental results](#). You know, *science!* But how do they apply to me? First, I established where the fold actually was. Per Google Analytics, **about 25 percent of our users are using screen resolutions where the page fold is at about 700 or 800 pixels of height**. And remember, browsers have a lot of horizontal chrome that tends to squander that height — toolbars, status bars, tabs, etc. The fold is probably much closer than you think it is.

Next, I looked at the advice I had been given regarding the top of the page. Sure enough, we had a bunch of irrelevant UI at the top that didn't really matter: things like redundant page titles, and two line title entry. **We were wasting critical real estate at the top of the page!** For the 25 percent of users who have a 700 or 800 pixel fold, items were pushed down far enough that they might not actually be visible. Worse still, the strong bottom border of the text entry area with the drag slider *could* possibly align with the page fold itself — leading the user to believe that nothing is below there and failing to scroll.

It's not only a basic rule of writing, it's also a basic rule of the web: put the most important content at as close to the top of the page as you can. This isn't new advice, but it's so important that it never hurts to revisit it periodically in your own designs.

In treating user myopia, it's not enough to place important stuff directly in the user's eyepoint. You also need to ensure that you've placed the absolute *most important stuff* at the top of the page — and haven't created any accidental barriers to scrolling, so they can find the rest of it. The fold is far less important than it used to be, but it isn't quite as mythical as Bigfoot and the Loch Ness Monster quite yet.

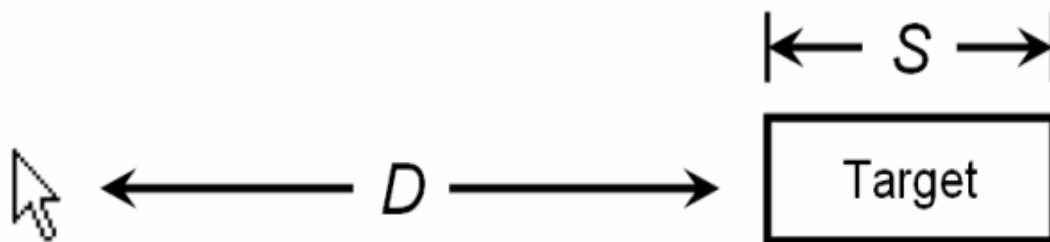
Fitts' Law and Infinite Width

[Fitts' Law](#) is arguably the most important formula in the field of human-computer interaction.

It's..

Time = $a + b \log_2 (D / S + 1)$

.. where D is the distance from the starting point of the cursor, and S is the width of the target. This is all considered on a 2D plane relative to the axis of movement.



Years of experimental results have [proven Fitts' law time and time again](#):

Fitts' law has been shown to apply under a variety of conditions, with many different limbs (hands, feet, head-mounted sights, eye gaze), manipulanda (input devices), physical environments (including underwater!), and user populations (young, old, mentally retarded, and drugged participants). Note that the constants a and b have different values under each of these conditions.

It's not exactly rocket science, as [Bruce Tognazzini points out](#):

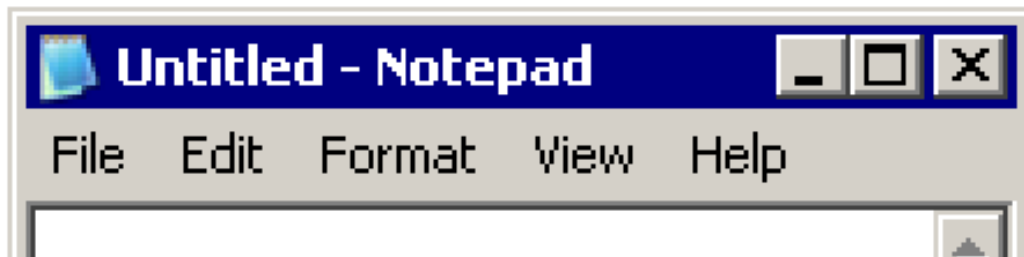
The time to acquire a target is a function of the distance to and size of the target.

While at first glance, this law might seem patently obvious, it is one of the most ignored principles in design. Fitts' law (properly, but rarely, spelled "Fitts' Law") dictates the Macintosh pull-down menu acquisition should be approximately five times faster than Windows menu acquisition, and this is proven out.

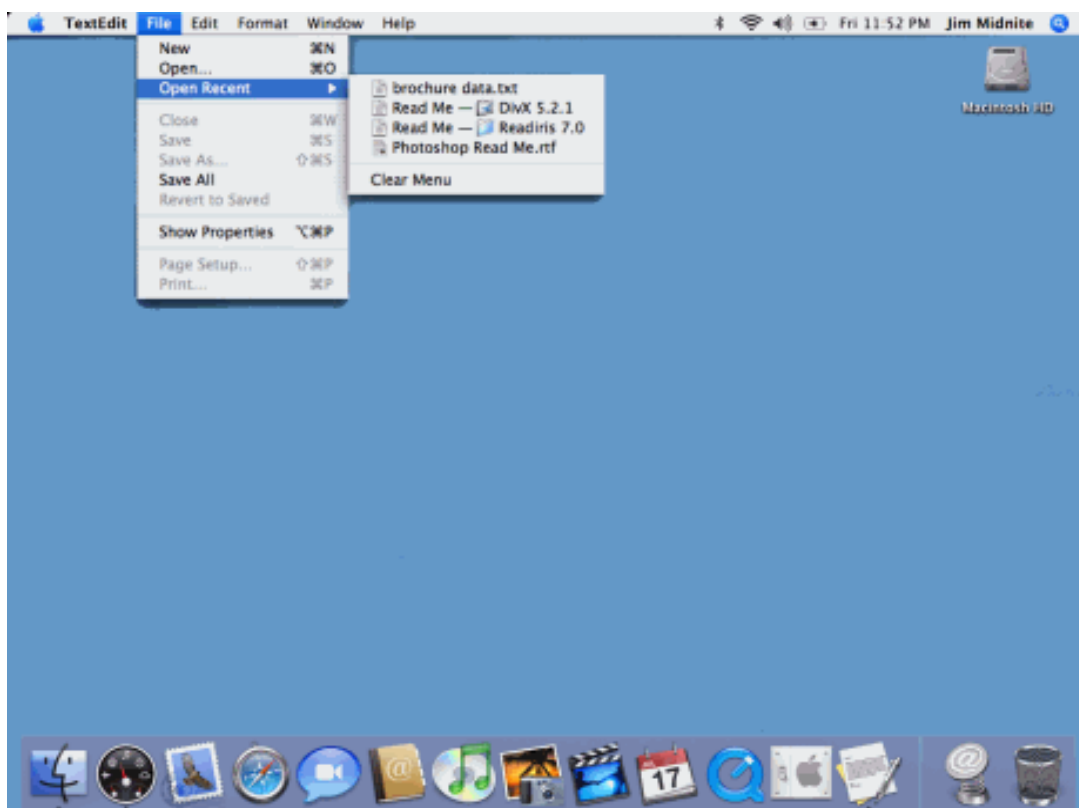
So, to make navigation easier, you either put clickable items closer together, or you

make the clickable area bigger. Or both. I know what you're thinking: *no duh*. But bear with me.

Here's one thing that puzzled me. I hate Windows as much as the next disestablishmentarianist, but how can the menu argument be valid? Are Macintosh pull-down menus really that much larger than Windows pull-down menus?



They aren't significantly larger. **But Macintosh menus aren't attached to the application window — they're always at the top of the screen.**



Since the cursor stops at the edge of the screen, **for the purposes of Fitts' law calculation, Macintosh menus are infinitely tall!** Thus, Macintosh menus *are* faster to navigate.

Although placing the menus at the top of the display does leverage Fitts' law nicely, it also presents its own set of problems.

- Where does the menu go in a multiple monitor scenario? [I use three monitors](#) on both my home and work PCs. If I move an application to the rightmost monitor, do the application menus still appear on the center or left monitor?
- Detaching applications from their UI in this manner seems to violate the rule of proximity — related things should be together. On a single monitor system, the distance between the application and its menu could be quite large unless the application window is maximized.
- In a broader sense, [I think the days of the main menu are numbered as a keystone GUI metaphor](#). As far back as I can remember, the Macintosh has always used this “menu at the top of the display” metaphor, so it’s written in stone for users at this point. Change could be painful. But then again, Apple has a habit of reinventing themselves periodically, so who knows.

Fitts’ law isn’t just about making things larger and easier to click on. It’s about [maximizing the utility of the natural borders on the edges of your screen](#):

Fitts’ law indicates that the most quickly accessed targets on any computer display are the four corners of the screen, because of their pinning action, and yet, for years, they seemed to be avoided at all costs by designers.

Use the pinning actions of the sides, bottom, top, and corners of your display: A single-row toolbar with tool icons that “bleed” into the edges of the display will be many times faster than a double row of icons with a carefully-applied one-pixel non-clickable edge between the tools and the side of the display.

I’ve definitely felt the pain of Fitts’ law violations.

I love multiple monitors. In my opinion, life begins with two displays, the largest you can afford. And you should really upsize to three if you want maximum benefit. But **one unfortunate side-effect of multiple monitors is the removal of some natural edges between adjoining monitors**. The cursor now flows freely between monitors; it’s painful to stop the cursor on the left and right edges of the app on the center monitor.

And Fitts’ law violations can also extend to hardware. Consider touchpad designs that have dedicated scrolling areas on the left or bottom.



This seems like a good idea on paper, but in practice, it destroys the usability of the touchpad. **On a touchpad with dedicated scrolling areas, you have no way to know when you've passed from touchpad area into the no-man's-land of scrolling area.** The natural edges of the touchpad are ruined; we've given them an arbitrarily different, hard-coded set of functionality. Dedicated hardware isn't even necessary to achieve scrolling effects on a touchpad. We can easily leverage Fitts' Law in the touchpad driver software instead. Just slide your finger until you hit an edge, then slide it along the edge.

The edges could be your most valuable real estate. Use them responsibly. Fitts' law is powerful stuff.

The Ultimate Unit Test Failure

We programmers are [obsessive by nature](#). But I often get frustrated with the depth of our obsession over things like code coverage. Unit testing and code coverage are [good things](#). But perfectly executed code coverage doesn't mean users will use your program. Or that it's even *worth* using in the first place. **When users can't figure out how to use your app, when users [pass over your app](#) in favor of something easier or simpler to use, that's the ultimate unit test failure.** *That's* the problem you should be trying to solve.

I want to run up to my fellow programmers and physically shake them: think bigger!

A perfect example of *thinking bigger* is Alan Cooper's [Interaction 08](#) keynote, **An Insurgency of Quality**.



There's [a transcript of his keynote available](#), or you can [view a video of his keynote video](#) with the slides in place.

Alan is a well-known [interaction designer](#), and the author of several classic books in the field, such as [About Face](#), and a few others that are on my [recommended reading list](#). In the Q&A after the talk, he had this to say:

“We are not very important because we don’t cut code.” (A boo and hiss from the audience.) In the world of high-technology, if you cut code, you control things. It’s the power to destroy the spice, it’s the power to control the spice. It’s not a fine kind of control: it’s bruce-force kind of things. [Interaction designers are] largely marginalized. We’re constantly asking for permission from the folks who shouldn’t be in a position to grant permission. We should be working with business folks and marshalling the technology to meet the solutions to business problems.

But when it comes time to marshal the solution to the problems, we find ourselves slamming into this kind of Stay-Puft Marshmallow Man of software development.



We don’t need to change interaction design; we need to re-orient organizations to build things right. When we come to programmers and say, “Look at the people I’ve talked to; look at the personas I’ve created” and present them with research, programmers understand that, and that’s how we will influence.

It pains me to hear that **Cooper considers most programmers twenty-story marshmallow barriers to good interaction design**. Please don’t be one of those programmers. Learn about the science of interaction design. Pick up a copy of [Don’t Make Me Think](#) as an introduction if you haven’t already. There’s a reason this book is at the top of my recommended reading list. Keep your unit testing and code coverage in

perspective — the *ultimate* unit test is whether or not users want to use your application. All the other tests you write are totally irrelevant until you can get that one to pass.

Coding Horror commentor Joeri Sebrechts:

Since it is often the user who notices a defect, it also doesn't matter whether something is a bug or a feature according to the developer. If the user perceives something to be a defect, it is a defect, whether in user training, user documentation, user interface or actual functionality.

Oct 13 '08 at 9:13

Version 1 Sucks, But Ship it Anyway

I've been unhappy with every single piece of software I've ever released. Partly because, like many software developers, I'm a perfectionist. And then, there are inevitably *problems*:

- The schedule was too aggressive and too short. We need more time!
- We ran into unforeseen technical problems that forced us to make compromises we are uncomfortable with.
- We had the wrong design, and needed to change it in the middle of development.
- Our team experienced internal friction between team members that we didn't anticipate.
- The customers weren't who we thought they were.
- Communication between the designers, developers and project team wasn't as efficient as we thought it would be.
- We overestimated how quickly we could learn a new technology.

The list goes on and on. Reasons for failure on a software project [are legion](#).

At the end of the development cycle, you end up with **software that is a pale shadow of the shining, glorious monument to software engineering that you envisioned when you started**.

It's tempting, at this point, to throw in the towel — to add more time to the schedule so you can get it right before shipping your software. Because, after all, [real developers ship](#).

I'm here to tell you that **this is a mistake**.

Yes, you did a ton of things wrong on this project. But you also did a ton of things wrong

that *you don't know about yet*. And there's no other way to find out what those things are until you ship this version and get it in front of users and customers. I think [Donald Rumsfeld put it best](#):

As we know,

There are known knowns.

There are things we know we know.

We also know

There are known unknowns.

That is to say

We know there are some things

We do not know.

But there are also unknown unknowns,

The ones we don't know

We don't know.

In the face of the inevitable end-of-project blues — rife with compromises and totally unsatisfying quick fixes and partial solutions — you could hunker down and lick your wounds. You could regroup and spend a few extra months fixing up this version before releasing it. You might even feel good about yourself for making the hard call to get the engineering right before unleashing yet another buggy, incomplete chunk of software on the world.

Unfortunately, this is an even bigger mistake than shipping a flawed version.

Instead of spending three months fixing up this version in a sterile, isolated lab, you *could* be spending that same three month period **listening to feedback from real live, honest-to-god, annoyingly dedicated users of your software**. Not the software as you imagined it, and the users as you imagined them, but as they exist in the real world. You can turn around and use that directed, real-world feedback to not only *fix* all the sucky parts of version 1, but spend your whole development budget more efficiently, predicated on hard usage data from your users.

Now, I'm not saying you should release crap. Believe me, we're all perfectionists here.

But the real world can be a cruel, unforgiving place for us perfectionists. It's saner to let go and realize that when your software crashes on the rocky shore of the real world, disappointment is inevitable, *but fixable!* What's important isn't so much the initial state of the software — in fact, some say [if you aren't embarrassed by v1.0 you didn't release it early enough](#) — but what you do *after* releasing the software.

The velocity and responsiveness of your team to user feedback will set the tone for your software, far more than any single release ever could. That's what you need to get good at. Not the platonic ideal of shipping mythical, perfect software, but being responsive to your users, to your customers, and demonstrating that through the act of continually improving and refining your software based on their feedback. So to the extent that you're optimizing for near-perfect software releases, you're optimizing for the wrong thing.

There's no *question* that, for whatever time budget you have, you will end up with better software by releasing as early as practically possible, and then spending the rest of your time [iterating rapidly based on real-world feedback](#).

So trust me on this one: **even if version 1 sucks, ship it anyway.**

Gift this book to your friends...for [free](#).

Become a Hyperink reader. Get a [special surprise](#).

Like the book? Support our author and leave a [comment!](#)

VIII.

Security Basics: Protecting Your Users' Data



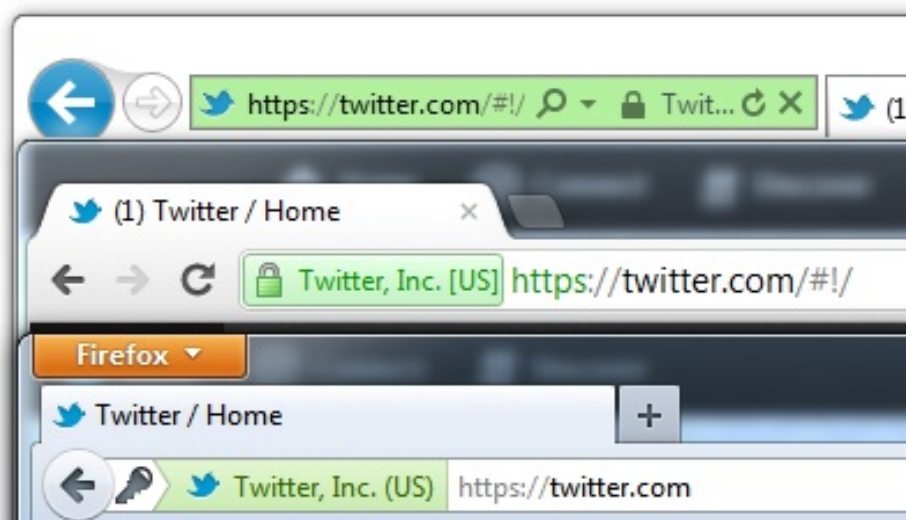
Should All Web Traffic Be Encrypted?

“...the more web sites you visit, the more networks you touch and trust with a username and password combination — the greater the odds that at least *one* of those networks will be compromised...”

The prevalence of free, open WiFi has made it **rather easy for a WiFi eavesdropper to steal your identity cookie for the websites you visit while you’re connected to that WiFi access point**. This is something I talked about in [Breaking the Web’s Cookie Jar](#). It’s difficult to fix without making major changes to the web’s infrastructure.

In the year since I wrote that, a number of major websites have “solved” the WiFi eavesdropping problem by either making encrypted HTTPS web traffic an *account option* or *mandatory* for all logged in users.

For example, I just noticed that Twitter, transparently to me and presumably all other Twitter users, **switched to an encrypted web connection by default**. You can tell because most modern browsers show the address bar in green when the connection is encrypted.



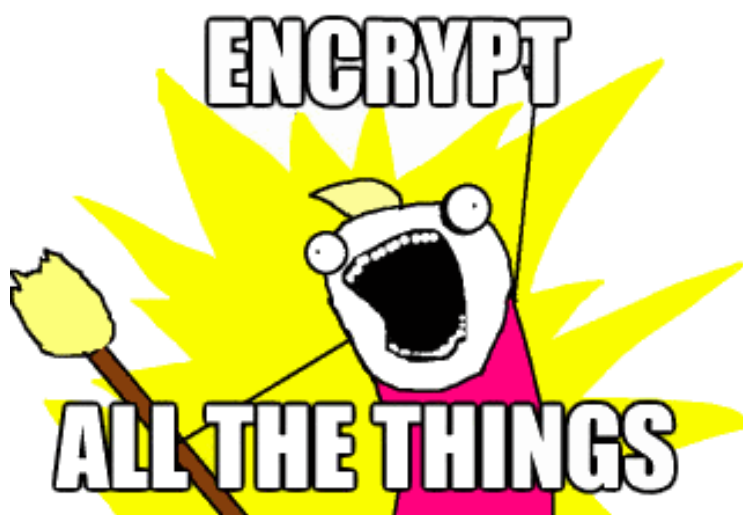
I initially resisted this as overkill, except for obvious targets like email (the [skeleton key to](#)

[all your online logins](#)) and banking.

Yes, **you can naively argue that every website should encrypt all their traffic all the time**, but to me that's a "boil the sea" solution. I'd rather see a better, more secure identity protocol than ye olde HTTP cookies. I don't actually care if anyone sees the rest of my public activity on Stack Overflow; it's hardly a secret. But gee, I sure *do* care if they [somehow sniff out my cookie and start running around doing stuff as me!](#) Encrypting everything just to protect that one lousy cookie header seems like a whole lot of overkill to me.

Of course, there's no reason to encrypt traffic for anonymous, not-logged-in users, and Twitter doesn't. You get a plain old HTTP connection until you log in, at which point they automatically switch to HTTPS encryption. Makes sense.

It was totally painless for me, as a user, and it makes stealing my Twitter identity, or [eavesdropping on my Twitter activity](#) (as fascinating as I know that must sound), dramatically more difficult. I can't really construct a credible argument *against* doing this, even for something as relatively trivial as my Twitter account, and it has some definite benefits. So perhaps Twitter has the right idea here; **maybe encrypted connections should be the default for all web sites**. As tinfoil hat as this seemed to me a year ago, now I'm wondering if that might actually be the right thing to do for the long-term health of the overall web, too.



Why not boil the sea, then? Let us *encrypt all the things!*

HTTPS isn't (that) expensive any more

Yes, in the hoary old days of the 1999 web, HTTPS was quite computationally expensive.

But thanks to 13 years of Moore's Law, that's no longer the case. It's still *more work* to set up, yes, but [consider the real world case of Gmail](#):

In January this year (2010), Gmail switched to using HTTPS for everything by default. Previously it had been introduced as an option, but now all of our users use HTTPS to secure their email between their browsers and Google, all the time. In order to do this we had to deploy no additional machines and no special hardware. On our production frontend machines, SSL/TLS accounts for less than 1 percent of the CPU load, less than 10KB of memory per connection and less than 2 percent of network overhead. Many people believe that SSL takes a lot of CPU time and we hope the above numbers (public for the first time) will help to dispel that.

HTTPS means The Man can't spy on your Internet

Since all the traffic between you and the websites you log in to would now be encrypted, the ability of nefarious evildoers to either ...

- steal your identity cookie
- peek at what you're doing
- see what you've typed
- interfere with the content you send and receive

... is, if not [completely eliminated](#), drastically limited. Regardless of whether you're on open public WiFi or not.

Personally, I don't care too much if people see what I'm doing online since the whole point of a lot of what I do is to ... [let people see what I'm doing online](#). But I certainly don't subscribe to the dangerous idea that "only criminals have things to hide." Everyone deserves the right to personal privacy. And there are lots of repressive governments out there who wouldn't hesitate at the chance to spy on what their citizens do online, or worse. Much, much worse. Why not improve the Internet for all of them at once?

HTTPS goes faster now

Security always comes at a cost, and encrypting a web connection is no different. HTTPS is going to be inevitably slower than a regular HTTP connection. But how *much* slower? It used to be that encrypted content wouldn't be cached in some browsers, but [that's no longer true](#). And Google's SPDY protocol, intended as a drop-in replacement for HTTP, even goes so far as to [bake encryption in by default](#), and not just for better performance:

It is a specific technical goal of SPDY to] make SSL the underlying transport protocol, for better security and compatibility with existing network infrastructure. Although SSL does introduce a latency penalty, we believe that the long-term future of the web depends on a secure network connection. In addition, the use of SSL is necessary to ensure that communication across existing proxies is not broken.

There's also [SSL False Start](#) which requires a modern browser, but reduces the [painful latency](#) inherent in the expensive, but necessary, handshaking required to get encryption going. SSL encryption of HTTP will never be *free*, exactly, but it's certainly a lot faster than it used to be, and getting faster every year.

Bolting on encryption for logged-in users is by no means an easy thing to accomplish, particularly on large, established websites. You won't see me out there berating every public website for not offering encrypted connections yesterday because I know how much work it takes, and how much additional complexity it can add to an already busy team. Even though HTTPS is way easier now than it was even a few years ago, there are still plenty of tough gotchas: proxy caching, for example, becomes vastly harder when the proxies can no longer "see" what the encrypted traffic they are proxying is doing. Most sites these days are a broad mashup of content from different sources, and technically *all* of them need to be on HTTPS for a properly encrypted connection. Relatively underpowered and weakly connected mobile devices will pay a much steeper penalty, too.

Maybe not tomorrow, maybe not next year, but over the medium to long term, **adopting encrypted web connections as a standard for logged-in users** is the healthiest direction for the future of the web. We need to work toward making HTTPS easier, faster and most of all, *the default* for logged in users.

Dictionary Attacks 101

Several high profile Twitter accounts [were hijacked](#) in 2009:

An 18-year-old hacker with a history of celebrity pranks has admitted to Monday's hijacking of multiple high-profile Twitter accounts, including President-Elect Barack Obama's, and the official feed for Fox News.

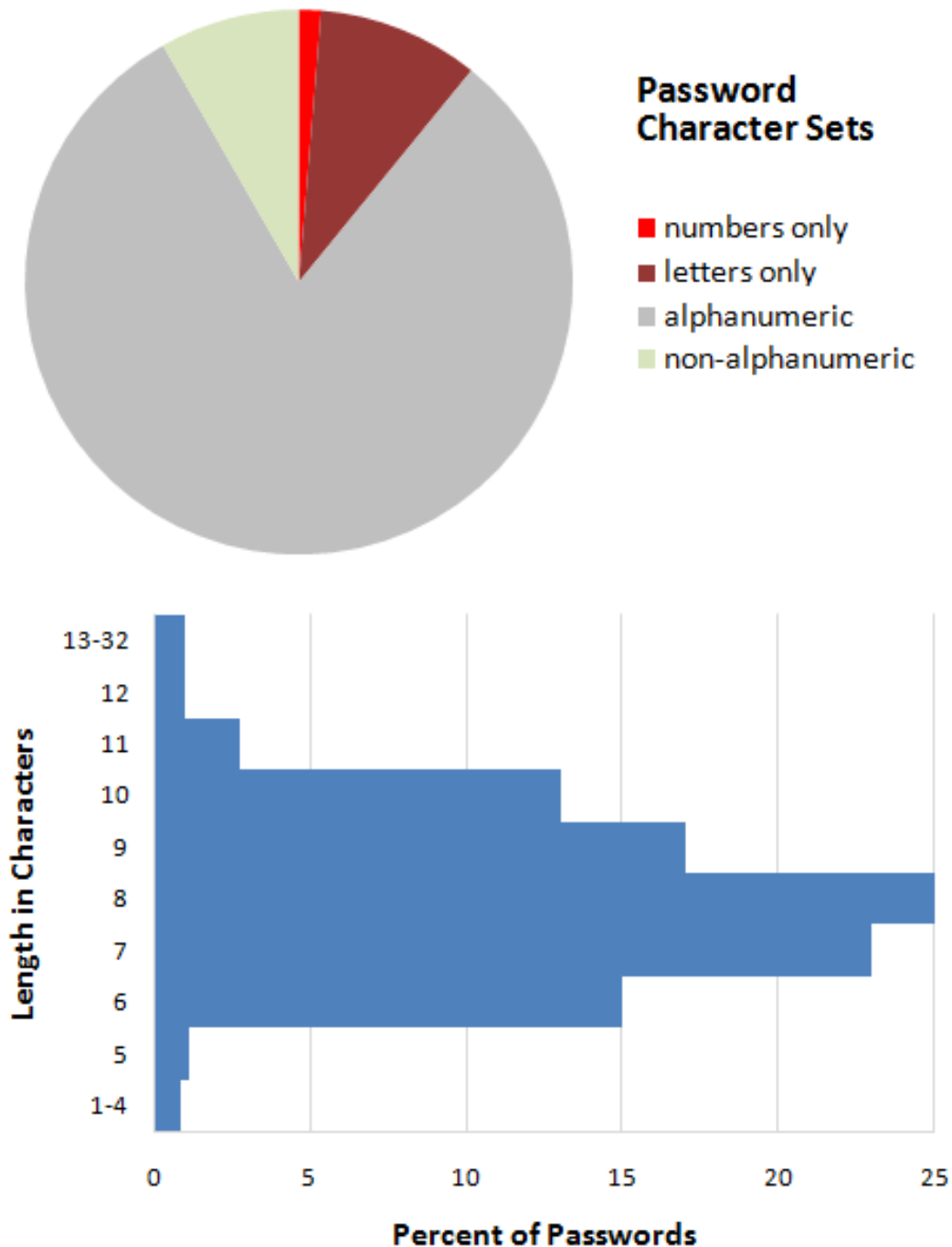
The hacker, who goes by the handle GMZ, told Threat Level on Tuesday he gained entry to Twitter's administrative control panel by pointing an automated password-guesser at a popular user's account. The user turned out to be a member of Twitter's support staff, who'd chosen the weak password "happiness."

Cracking the site was easy, because Twitter allowed an unlimited number of rapid-fire log-in attempts.

"I feel it's another case of administrators not putting forth effort toward one of the most obvious and overused security flaws," he wrote in an IM interview. "I'm sure they find it difficult to admit it."

If you're a moderator or administrator it is *especially* negligent to have such an easily guessed password. But the real issue here is the way Twitter allowed unlimited, as-fast-as-possible login attempts.

Given the **average user's password choices** — as documented by Bruce Schneier's [analysis of 34,000 actual MySpace passwords](#) captured from a [phishing attack](#) in late 2006 — this is a pretty scary scenario.



Based on this data, the average MySpace user has an 8-character alphanumeric password. Which isn't great, but doesn't sound *too* bad. That is, until you find out that 28 percent of those alphanumerics were all lowercase with a single final digit — and two-thirds of the time that final digit was 1!

Yes, [brute force attacks are still for dummies](#). Even the typically terrible MySpace password — eight character all lowercase, ending in 1, would require around 8 billion login attempts:

$26 \times 26 \times 26 \times 26 \times 26 \times 26 \times 26 \times 1 = 8,031,810,176$

At one attempt per second, that would take more than 250 years. *Per user!*

But a [dictionary attack](#), like the one used in the Twitter hack? Well, that's another story. The entire Oxford English Dictionary [contains around 171,000](#) words. As you might imagine, the average person only uses a tiny fraction of those words, by some estimates [somewhere between 10 and 40 thousand](#). At one attempt per second, we could try **every word in the Oxford English Dictionary in slightly less than two days**.

Clearly, the *last* thing you want to do is give attackers carte blanche to run unlimited login attempts. All it takes is one user with a weak password to provide attackers a toehold in your system. In Twitter's case, the attackers really hit the jackpot: the user with the weakest password happened to be a member of the Twitter administrative staff.

Limiting the number of login attempts per user is security 101. If you don't do this, you're practically setting out a welcome mat for anyone to launch a dictionary attack on your site, an attack that gets statistically more effective every day the more users you attract. In some systems, your account can get locked out if you try and fail to log in a certain number of times in a row. This can lead to denial of service attacks, however, and is generally discouraged. It's more typical for each failed login attempt to take longer and longer, like so:

1st failed login	no delay
2nd failed login	2 sec delay
3rd failed login	4 sec delay
4th failed login	8 sec delay
5th failed login	16 sec delay

And so on. Alternately, you could display a [CAPTCHA](#) after the fourth attempt.

There are endless variations of this technique, but the net effect is the same: attackers can only try a handful of passwords each day. A brute force attack is out of the question, and a broad dictionary attack becomes impractical, at least in any kind of human time.

It's tempting to blame Twitter here, but honestly, I'm not sure they're alone. I [forget my passwords a lot](#). I've made at least five or six attempts to guess my password on multiple websites and I can't recall ever experiencing any sort of calculated delay or account

lockouts.

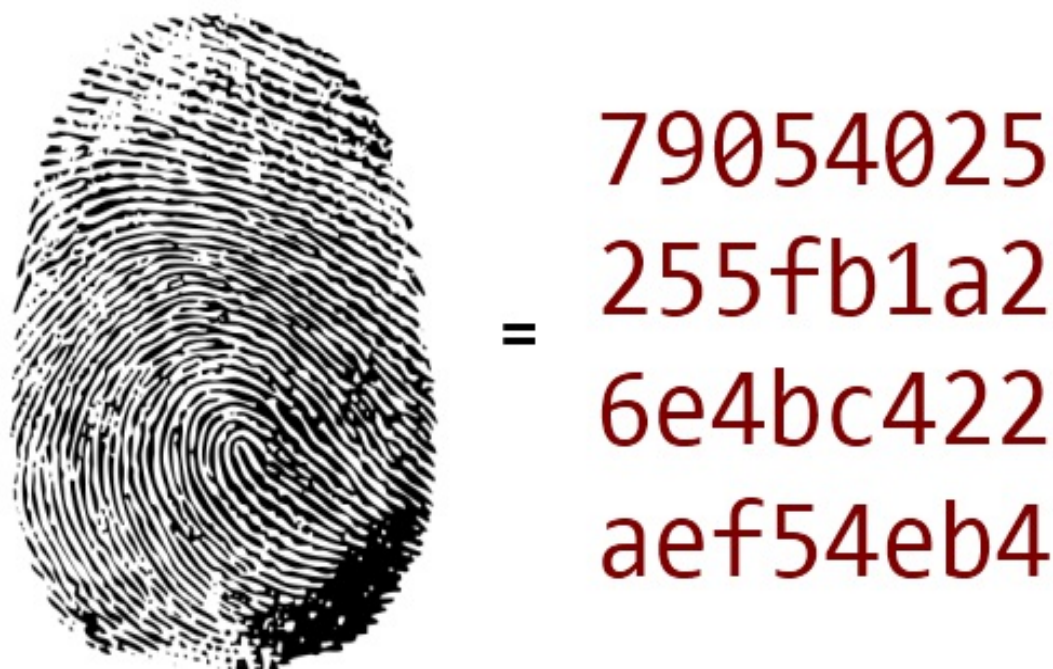
I'm reasonably sure the big commercial sites have this mostly figured out. But since [every rinky-dink website on the planet demands that I create unique credentials especially for them](#), any of them could be vulnerable. **You better hope they're all smart enough to throttle failed logins** — and that you're careful to use unique credentials on every single website you visit.

Maybe this was less of a problem in the [bad old days of modems](#), as there were severe physical limits on how fast data could be transmitted to a website, and how quickly that website could respond. But today, we have the one-two punch of naive websites running on blazing fast hardware, and users with speedy broadband connections. Under these conditions, I could see attackers regularly achieving up to two password attempts per second.

If you thought of dictionary attacks as mostly a desktop phenomenon, perhaps it's time to revisit that assumption. As Twitter illustrates, the web now offers ripe conditions for dictionary attacks. I urge you to test your website, or any websites you use — and **make sure they all have some form of failed login throttling in place.**

Speed Hashing

Hashes are a bit like fingerprints for data.



A given hash uniquely represents a file or any arbitrary collection of data. At least [in theory](#). This is a 128-bit MD5 hash you're looking at above, so it can represent at most 2,128 unique items, or 340 trillion trillion *trillion*. In reality the usable space is substantially less; you can start seeing significant collisions once you've filled [half the square root of the space](#), but the square root of an impossibly large number is still impossibly large.

Back in 2005, I wondered about the difference between a [checksum and a hash](#). You can think of a checksum as a person's full name: **Eubediah Q. Horsefeathers**. It's a shortcut to uniqueness that's fast and simple, but easy to forge, because security isn't really the point of naming. You don't walk up to someone and demand their fingerprints to prove they are who they say they are. Names are just convenient disambiguators, a way of quickly determining who you're talking to for social reasons, not absolute proof of identity. There can certainly be multiple people in the world with the same name, and it wouldn't be too much trouble to legally change your name to match someone else's. But changing your *fingerprint* to match Eubediah's is another matter entirely; that should be

impossible except [in the movies](#).

Secure hashes are designed to be tamper-proof.

A properly designed secure hash function **changes its output radically with tiny single bit changes to the input data**, even if those changes are malicious and intended to cheat the hash. Unfortunately, not all hashes were designed properly, and some, like MD5, [are outright broken and should probably be reverted to checksums](#).

As we will explain below, the algorithm of Wang and Yu can be used to create files of arbitrary length that have identical MD5 hashes, and that differ only in 128 bytes somewhere in the middle of the file. Several people have used this technique to create pairs of interesting files with identical MD5 hashes:

- Magnus Daum and Stefan Lucks have created [two PostScript files with identical MD5 hash](#), of which one is a letter of recommendation, and the other is a security clearance.
- Eduardo Diaz has described a [scheme](#) by which two programs could be packed into two archives with identical MD5 hash. A special “extractor” program turn one archive into a “good” program and the other into an “evil” one.
- In 2007, Marc Stevens, Arjen K. Lenstra, and Benne de Weger used an improved version of Wang and Yu’s attack known as the [chosen prefix collision](#) method to produce two executable files with the same MD5 hash, but different behaviors. Unlike the old method, where the two files could only differ in a few carefully chosen bits, the chosen prefix method allows two completely arbitrary files to have the same MD5 hash, by appending a few thousand bytes at the end of each file.
- Didier Stevens used the evilize program (below) to create [two different programs with the same Authenticode digital signature](#). Authenticode is Microsoft’s code signing mechanism, and although it uses SHA1 by default, it still supports MD5.

If you could mimic another person’s fingerprint or DNA at will, you could do some *seriously* evil stuff. MD5 is clearly compromised, and SHA-1 is [not looking too great](#) these days.

The good news is that hashing algorithms (assuming you didn’t roll your own, God forbid) were designed by professional mathematicians and cryptographers who knew what they were doing. Just pick a hash of a newer vintage than MD5 (1991) and SHA-1 (1995), and you’ll be fine — at least as far as collisions and uniqueness are concerned. But keep

reading.

Secure hashes are designed to be slow

Speed of a checksum calculation is important, as checksums are generally working on data as it is being transmitted. If the checksum takes too long, it can affect your transfer speeds. If the checksum incurs significant CPU overhead, that means transferring data will also slow down or overload your PC. For example, imagine the sort of checksums that are used on video standards like [DisplayPort](#), which can peak at 17.28 Gbit/sec.

But hashes aren't designed for speed. In fact, quite the opposite: **hashes, when used for security, need to be slow**. The faster you can calculate the hash, the more viable it is to use brute force to mount attacks. Unfortunately, "slow" in 1990 and 2000 terms may not be enough. The hashing algorithm designers may have anticipated predicted increases in CPU power via Moore's Law, but they almost certainly did *not* see the radical increases in GPU computing power coming.

How radical? Well, compare the results of CPU powered [hashcat](#) with the GPU powered [oclHashcat](#) when calculating MD5 hashes:

Radeon 7970	8213.6 M c/s
6-core AMD CPU	52.9 M c/s

The GPU on a single modern video card produces **over 150 times the number of hash calculations per second** compared to a modern CPU. If Moore's Law anticipates a [doubling of computing power every 18 months](#), that's like peeking **10 years into the future**. Pretty amazing stuff, isn't it?

Hashes and passwords

Let's talk about passwords, since hashing and passwords are intimately related. Unless [you're storing passwords incorrectly](#), you *always* store a user's password as a salted hash, [never as plain text](#). Right? *Right?* This means if your database containing all those hashes is [compromised or leaked](#), the users are still protected — nobody can figure out what their password actually is based on the hash stored in the database. Yes, there are of course [dictionary attacks that can be surprisingly effective](#), but we can't protect users dead-set on using "monkey1" for their password from themselves. And anyway, the real solution to users choosing crappy passwords is not to make users remember ever more complicated and longer passwords, but to [do away with passwords altogether](#).

This has one unfortunate ramification for password hashes: very few of them were

run. This will also give you some idea how computationally expensive various known hashes are on GPUs relative to each other, such as:

MD5	23070.7 M/s
SHA-1	7973.8 M/s
SHA-256	3110.2 M/s
SHA-512	267.1 M/s
NTLM	44035.3 M/s
DES	185.1 M/s
WPA/WPA2	348.0 k/s

What about rainbow tables?

Rainbow tables are [huge pre-computed lists of hashes](#), trading off table lookups to massive amounts of disk space (and potentially memory) for raw calculation speed. They are now utterly and completely obsolete. Nobody who knows what they're doing would bother. They'd be wasting their time. I'll let [Coda Hale explain](#):

Rainbow tables, despite their recent popularity as a subject of blog posts, have not aged gracefully. Implementations of password crackers can leverage the massive amount of parallelism available in GPUs, peaking at billions of candidate passwords a second. You can literally test all lowercase, alphabetic passwords which are ≤ 7 characters in less than 2 seconds. And you can now rent the hardware which makes this possible to the tune of less than \$3/hour. For about \$300/hour, you could crack around 500,000,000,000 candidate passwords a second.

Given this massive shift in the economics of cryptographic attacks, it simply doesn't make sense for anyone to waste terabytes of disk space in the hope that their victim didn't use a salt. It's a lot easier to just crack the passwords. Even a "good" hashing scheme of SHA256(salt + password) is still completely vulnerable to these cheap and effective attacks.

But when I store passwords I use salts so none of this applies to me!

Hey, awesome, you're smart enough to not just use a hash, but also to [salt the hash](#). Congratulations.

```
$saltedpassword = sha1(SALT . $password);
```

I know what you're thinking. "I can hide the salt, so the attacker won't know it!" You can

certainly try. You could put the salt somewhere else, like in a different database, or put it in a configuration file, or in some hypothetically secure hardware that has additional layers of protection. In the event that an attacker obtains your database with the password hashes, but somehow has no access to or knowledge of the salt it's theoretically possible.

This will provide the illusion of security more than any actual security. Since you need both the salt and the choice of hash algorithm to generate the hash, and to check the hash, it's unlikely an attacker would have one but not the other. If you've been compromised to the point that an attacker has your password database, it's reasonable to assume they either have or can get your secret, hidden salt.

The first rule of security is to always assume and plan for the worst. Should you use a salt, ideally a random salt for each user? Sure, it's definitely a good practice, and at the very least it lets you disambiguate two users who have the same password. But these days, **salts alone can no longer save you** from a person willing to spend a few thousand dollars on video card hardware, and if you think they can, you're in trouble.

I'm too busy to read all this.

If you are a user:

Make sure all your passwords are 12 characters or more, ideally a lot more. [I recommend adopting pass phrases](#), which are not only a lot easier to remember than passwords (if not type) but also *ridiculously* secure against brute forcing purely due to their length.

If you are a developer:

Use [bcrypt](#) or [PBKDF2](#) exclusively to hash anything you need to be secure. These new hashes were specifically designed to be [difficult to implement on GPUs](#). Do *not* use any other form of hash. Almost every other popular hashing scheme is vulnerable to brute forcing by arrays of commodity GPUs, which only get faster and more parallel and easier to program for every year.

The Dirty Truth About Web Passwords

In December 2010, **the Gawker network was compromised**, resulting in a security breach at Lifehacker, Gizmodo, Gawker, Jezebel, io9, Jalopnik, Kotaku, Deadspin and Fleshbot. If you're a commenter on any of our sites, you probably have several questions.

It's no [Black Sunday](#) or [iPod modem firmware hack](#), but it has *release notes* — and the story it tells is as epic as Beowulf:

So, here we are again with a monster release of ownage and data droppage. Previous attacks against the target were mocked, so we came along and raised the bar a little. How's this for "script kids"? Your empire has been compromised, your servers, your databases, online accounts and source code have all been ripped to shreds!

You wanted attention, well guess what, You've got it now!

[Read those release notes](#). It'll explain how the compromise unfolded, blow by blow, from the inside.

Gawker is operated by Nick Denton, notorious for the unapologetic and often unethical "publish whatever it takes to get traffic" methods endorsed on his network. Do you remember the [iPhone 4 leak](#)? That was Gawker. Do you remember the article about [bloggers being treated as virtual sweatshop workers](#)? That was Gawker. Do you remember hearing about a blog lawsuit? [That was probably Gawker, too](#).

Some might say having every account on your network compromised is *exactly* the kind of unwanted publicity attention that Gawker was founded on.

Personally, I'm more interested in **how we can learn from this hack**. Where did Gawker go wrong, and how can we avoid making those mistakes on *our* projects?

1. **Gawker saved passwords**. You should never, *ever* store user passwords. If you do, [you're storing passwords incorrectly](#). Always store the [salted hash](#) of the password — *never* the password itself! It's so easy, even [members of Mensa](#) er .. can't .. figure it out.

2. **Gawker used encryption incorrectly.** The odd choice of archaic DES encryption meant that the passwords they saved were all truncated to 8 characters. No matter how long your password actually *was*, you only had to enter the first 8 characters for it to work. So much for [choosing a secure pass phrase](#). Encryption is only as effective as the person using it. I'm not smart enough to use encryption, either, as you can see in [Why Isn't My Encryption.. Encrypting?](#)
3. **Gawker asked users to create a username and password on their site.** The FAQ they posted about the breach has [two interesting clarifications](#):

2) *What if I logged in using Facebook Connect? Was my password compromised?*

No. We never stored passwords of users who logged in using Facebook Connect.

3) *What if I linked my Twitter account with my Gawker Media account? Was my Twitter password compromised?*

No. We never stored Twitter passwords from users who linked their Twitter accounts with their Gawker Media account.

That's right, people who used their [internet driver's license](#) to authenticate on these sites *had no security problems at all!* Does the need to post a comment on Gizmodo really *justify* polluting the world with [yet another username and password](#)? It's only the poor users who decided to entrust Gawker with a unique username and 'secure' password who got compromised.

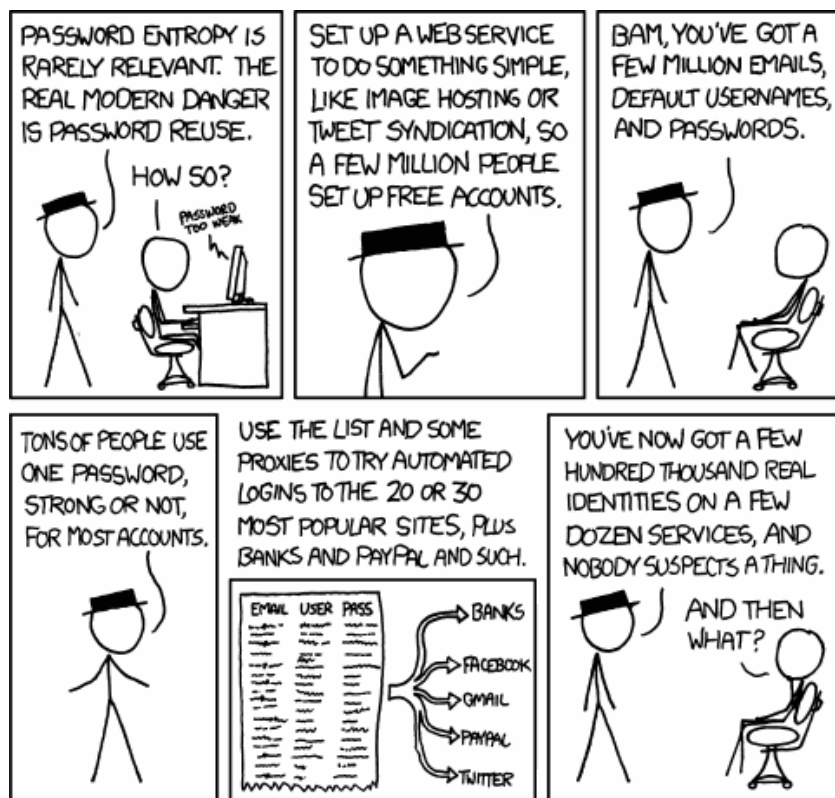
(Beyond that, "don't be a jerk" is good advice to follow in business as well as your personal life. I find that you generally get back what you give. When your corporate mission is to succeed by exploiting every quasi-legal trick in the book, surely you can't be surprised when you get the same treatment in return.)

But honestly, as much as we can point and laugh at Gawker and blame them for this debacle, there is absolutely nothing unique or surprising about any of this.

Here's the dirty truth about website passwords: the internet is full of websites exactly like the Gawker network. Let's say you have good old traditional username and passwords on 50 different websites. That's 50 different programmers who all have different ideas of how your password should be stored. I hope for your sake you used a different (and extremely secure) password on every single one of those websites. Because *statistically speaking, you're screwed.*

In other words, the more web sites you visit, the more networks you touch and trust with a username and password combination — the greater the odds that at least *one* of those networks will be **compromised exactly like Gawker was**, and give up your credentials for the world to see. At that point, unless you picked a strong, unique password on every single site you’ve ever visited, the situation gets ugly.

The bad news is that most users don’t pick strong passwords. This has been proven [time and time again](#), and the Gawker data is [no different](#). Even worse, most users re-use these bad passwords across multiple websites. That’s how [this ugly Twitter worm](#) suddenly appeared on the back of a bunch of compromised Gawker accounts.



Now do you understand why I’ve been so aggressive about promoting the concept of the [internet driver’s license](#)? That is, logging on to a web site using a set of **third party credentials from a company you can actually trust** to not be utterly incompetent at security? Sure, we’re centralizing risk here to, say, Google, or Facebook — but I trust Google a heck of a lot more than I trust J. Random Website, and this really is no different in practice than having password recovery emails sent to your GMail account.

I’m not here to criticize Gawker. On the contrary, I’d like to thank them for illustrating in broad, bold relief the dirty truth about website passwords: we’re all better off without them. If you’d like to see a future web free of Gawker style password compromises —

stop trusting every random internet site with a unique username and password! Demand that they allow you to use your internet driver's license — that is, your existing Twitter, Facebook, Google, or OpenID credentials — to log into their website.

Gift this book to your friends...for [free](#).

Become a Hyperink reader. Get a [special surprise](#).

Like the book? Support our author and leave a [comment](#)!

IX.

Testing Your Code, So it Doesn't Suck More Than it Has To



Sharing the Customer's Pain

“It’s incredibly important for developers to be in the trenches with the customers — the people who have to live with their code.”

In this [interview with Werner Vogels](#), the CTO of Amazon, he outlines how Amazon’s developers stay in touch with their users:

Remember that most of our developers are in the loop with customers, so they have a rather good understanding about what our customers like, what they do not like, and what is still missing.

We have a lot of feedback coming out of customer service. Many Amazonians have to spend some time with customer service every two years, actually listening to customer service calls, answering customer service e-mails, really understanding the impact of the kinds of things they do as technologists. This is extremely useful, because they begin to understand that our user base is not necessarily the techno-literate engineer. Rather, you may get a call from a grandma in front of a computer in a library who says she wants to buy something for her grandson who is at college and who has a Wishlist on Amazon.

Customer service statistics are also an early indicator if we are doing something wrong, or what the real pain points are for our customers. Sometimes in meetings we use a “voice of the customer,” which is a realistic story from a customer about some specific part of the Amazon experience. This helps managers and engineers to connect with the fact that we build many of these technologies for real people.

It’s easy to fall into [a pattern of ivory tower software development](#). All too often, software developers are merely tourists in their own codebase. Sure, they *write* the code, but they don’t *use it* on a regular basis like their customers do. They will visit from time to time, but they lack the perspective and understanding of users who — either by choice or by corporate mandate — live in that software as a part of their daily routine. As a result, problems and concerns are hard to communicate. They arrive as dimly heard messages from a faraway land.

When was the last time you even met a customer, much less tried to talk to them about a problem they’re having with your website or software?

It's incredibly important for developers to be in the trenches with the customers — the people who have to live with their code. Without a basic understanding of your users and customers, it's impossible to build [considerate software](#). And nothing builds perspective like being on the front lines of support. I'm not proposing that all programmers pull double duty as support. It'd be impossible to get any work done. But a brief rotation through support would do wonders for the resulting quality and usability of your software, exactly as described by Mr. Vogels.

[Dogfooding](#) can be difficult. But manning the support desk, if only for a short while, isn't. **Software developers should share the customer's pain.** I know it's not glamorous. But until you've demonstrated a willingness to help the customers using the software you've built — and more importantly, learn *why* they need help — you haven't truly finished building that software.

Working With the Chaos Monkey

Late last year, the Netflix Tech Blog wrote about [five lessons they learned moving to Amazon Web Services](#). AWS is, of course, the preeminent provider of so-called “cloud computing,” so this can essentially be read as **key advice for any website considering a move to the cloud**. And it’s great advice, too. Here’s the one bit that struck me as most essential:

We’ve sometimes referred to the Netflix software architecture in AWS as our Rambo Architecture. Each system has to be able to succeed, no matter what, even all on its own. We’re designing each distributed system to expect and tolerate failure from other systems on which it depends.

If our recommendations system is down, we degrade the quality of our responses to our customers, but we still respond. We’ll show popular titles instead of personalized picks. If our search system is intolerably slow, streaming should still work perfectly fine.

One of the first systems our engineers built in AWS is called the Chaos Monkey. The Chaos Monkey’s job is to randomly kill instances and services within our architecture. If we aren’t constantly testing our ability to succeed despite failure, then it isn’t likely to work when it matters most — in the event of an unexpected outage.

Which, let’s face it, seems like insane advice at first glance. I’m not sure many companies even understand why this would be a good idea, much less have the guts to attempt it. Raise your hand if where you work, *someone deployed a daemon or service that randomly kills servers and processes in your server farm*.

Now raise your other hand if that person is still employed by your company.

Who in their right mind would willingly choose to work with a Chaos Monkey?



Sometimes you don't get a choice; the Chaos Monkey chooses you. At [Stack Exchange](#), we struggled for months with a bizarre problem. **Every few days, one of the servers in the [Oregon web farm](#) would simply stop responding to all external network requests.** No reason, no rationale, and no recovery except for a slow, excruciating shutdown sequence requiring the server to bluescreen before it would reboot.

We spent months — literally *months* — chasing this [problem](#) down. We walked the list of everything we could think of to solve it, and then some:

- swapping network ports
- replacing network cables
- a different switch
- multiple versions of the network driver
- tweaking OS and driver level network settings
- simplifying our network configuration and removing [TProxy](#) for more traditional X-FORWARDED-FOR
- switching virtualization providers
- changing our [TCP/IP host model](#)
- getting Kernel hotfixes and applying them
- involving high-level vendor support teams

- some other stuff that I've now forgotten because I blacked out from the pain

At one point in this saga our team almost came to blows because we were so frustrated. (Well, as close to “blows” as a [remote team](#) can get over Skype, but you know what I mean.) Can you blame us? Every few days, one of our servers — no telling which one — would randomly wink off the network. **The Chaos Monkey strikes again!**

Even in our time of greatest frustration, I realized that there was a positive side to all this:

- Where we had one server performing an essential function, we switched to two.
- If we didn't have a sensible fallback for something, we created one.
- We removed dependencies all over the place, paring down to the absolute minimum we required to run.
- We implemented workarounds to stay running at all times, even when services we previously considered essential were suddenly no longer available.

Every week that went by, we made our system a tiny bit more redundant, because we had to. Despite the ongoing pain, it became clear that Chaos Monkey was actually doing us a big favor by forcing us to become extremely resilient. Not tomorrow, not someday, not at some indeterminate “we'll get to it eventually” point in the future, but *right now when it hurts*.

When you work with the Chaos Monkey, you quickly learn that everything happens for a reason. Except for those things which happen completely randomly. And that's why, even though it sounds crazy, **the best way to avoid failure is to fail constantly.**

Code Reviews: Just Do It

In [The Soft Side of Peer Reviews](#), Karl Wieggers starts with a powerful pronouncement:

Peer review — an activity in which people other than the author of a software deliverable examine it for defects and improvement opportunities — is one of the most powerful software quality tools available. Peer review methods include inspections, walkthroughs, peer deskchecks, and other similar activities. After experiencing the benefits of peer reviews for nearly fifteen years, I would never work in a team that did not perform them.

After participating in code reviews for a while here at Vertigo, I believe that **peer code reviews are the single biggest thing you can do to improve your code**. If you're not doing code reviews *right now* with another developer, you're missing a lot of bugs in your code and cheating yourself out of some key professional development opportunities. As far as I'm concerned, my code isn't done until I've gone over it with a fellow developer.

But don't take my word for it. McConnell provides plenty of evidence for the efficacy of code reviews in [Code Complete](#):

.. software testing alone has limited effectiveness — the average defect detection rate is only 25 percent for unit testing, 35 percent for function testing, and 45 percent for integration testing. In contrast, the average effectiveness of design and code inspections are 55 and 60 percent. Case studies of review results have been impressive:

- *In a software-maintenance organization, 55 percent of one-line maintenance changes were in error before code reviews were introduced. After reviews were introduced, only 2 percent of the changes were in error. When all changes were considered, 95 percent were correct the first time after reviews were introduced. Before reviews were introduced, under 20 percent were correct the first time.*
- *In a group of 11 programs developed by the same group of people, the first 5 were developed without reviews. The remaining 6 were developed with reviews. After all the programs were released to production, the first 5 had an average of 4.5 errors per 100 lines of code. The 6 that had been inspected had an average of only 0.82 errors per 100. Reviews cut the errors by over 80 percent.*

- *The Aetna Insurance Company found 82 percent of the errors in a program by using inspections and was able to decrease its development resources by 20 percent.*
- *IBM's 500,000 line Orbit project used 11 levels of inspections. It was delivered early and had only about 1 percent of the errors that would normally be expected.*
- *A study of an organization at AT&T with more than 200 people reported a 14 percent increase in productivity and a 90 percent decrease in defects after the organization introduced reviews.*
- *Jet Propulsion Laboratories estimates that it saves about \$25,000 per inspection by finding and fixing defects at an early stage.*

The only hurdle to a code review is finding a developer you respect to do it, and making the time to perform the review. Once you get started, I think you'll quickly find that every minute you spend in a code review is paid back tenfold.

If your organization is new to code reviews, I highly recommend Karl's book, [Peer Reviews in Software: A Practical Guide](#). The [sample chapters](#) Karl provides on his website are a great primer, too.

Testing With The Force

Markdown was one of the [humane markup languages](#) that we evaluated and adopted for Stack Overflow. I've been pretty happy with it, overall. So much so that I wanted to implement a tiny, lightweight subset of Markdown for comments as well.

I settled on these three commonly used elements:

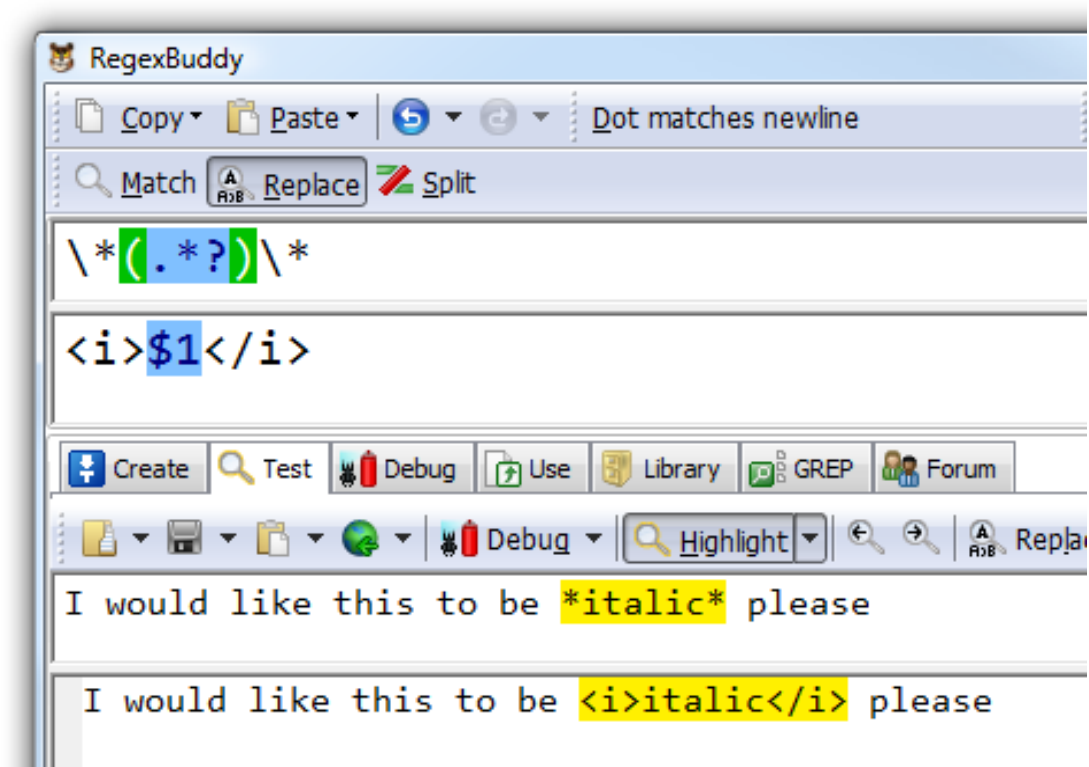
`*italic*` or `_italic_`

`**bold**` or `__bold__`

``code``

I [loves me some regular expressions](#) and this is exactly the stuff regex was born to do! It doesn't *look* very tough. So I dusted off [my copy of RegexBuddy](#) and began.

I typed some test data in the test window, and whipped up a little regex in no time at all. This isn't my first time at the disco.



Bam! Yes! Done and *done!* By gum, [I must be a genius programmer!](#)

Despite my obvious genius, I began to have some small, nagging doubts. Is the test phrase...

I would like this to be **italic** please.

... *really* enough testing?

Sure it is! I can feel in my bones that this thing freakin' works! It's almost like I'm being pulled toward shipping this code by some inexorable, dark, testing ... force. It's so *seductively easy!*



But wait. **I have this whole database of real world comments** that people have entered on Stack Overflow. Shouldn't I perhaps try my awesome regular expression on that corpus of data to see what happens? Oh, fine. If we must. Just to humor you, nagging doubt. Let's run a query and see.

```
select Text from PostComments
where dbo.RegexIsMatch(Text, '\*(.*?)\*') = 1
```

Which produced this list of matches, among others:

Interesting fact about math: $x * 7 == x + (x * 2) + (x * 4)$, or $x + x >> 1 + x >> 2$.
Integer addition is usually pretty cheap.

Thanks. What I needed was to turn on Singleline mode too, and use `.*?` instead of `.*`.

yeah, see my edit - change `select *` to `select RESULT.*` one row - are sure you have more than one row item with the same InstanceGUID?

Not your main problem, but you are mix and matching `wchar_t` and `TCHAR`. `mbstowcs()` converts from `char *` to `wchar_t *`.

aawwww.... Brainf**k is not valid. :/

Thank goodness I listened to my midichlorians and let **the light side of the testing force** prevail here!



So how do we fix this regex? We use the light side of the force — brute force, that is, against a ton of test cases! My job here is relatively easy because I have over 20,000 test cases sitting in a database. You may not have that luxury. Maybe you'll need to go out and find a bunch of test data on the internet somewhere. Or write a function that generates random strings to feed to the routine, also known as [fuzz testing](#).

I wanted to leave the rest of this regular expression as an exercise for the reader, as I'm a sick guy who finds that sort of thing entertaining. If you don't — well, what the heck is wrong with *you*, man? But I digress. I've been criticized for not providing, you know, “the answer” in my blog posts. Let's walk through some improvements to our italic regex pattern.

First, let's make sure we have **at least one non-whitespace character inside the asterisks**. And more than one character in total so we don't match the `**` case. We'll use [positive lookahead and lookbehind](#) to do that.

```
\*(?=\S)(.+?)(?<=\S)\*
```

That helps a lot, but we can test against our data to discover some other problems. We get into trouble when there are unexpected characters in front of or behind the asterisks,

like, say, `p*q*r`. So let's specify that **we only want certain characters outside the asterisks**.

```
(?<=[\s^,()]\*(?=\S)(.+?)(?<=\S)\*(?=[\s$,.,?!])
```

Run this third version against the data corpus, and wow, that's starting to look pretty darn good! There are undoubtedly some edge conditions, particularly since we're unlucky enough to be talking about code in a lot of our comments, which has wacky asterisk use.

This regex doesn't have to be (and probably *cannot* be, given the huge possible number of human inputs) perfect, but running it against a large set of input test data gives me reasonable confidence that I'm not totally screwing up.

So by all means, test your code with the force — brute force! It's good stuff! Just **be careful not to get sloppy, and let the dark side of the testing force prevail**. If you think one or two simple test cases covers it, that's taking the easy (and most likely, buggy and incorrect) way out.

I Pity the Fool Who Doesn't Write Unit Tests

J. Timothy King has a nice piece on [the twelve benefits of writing unit tests first](#).

Unfortunately, he seriously undermines his message by ending with this:

However, if you are one of the [coders who won't give up code-first], one of those curmudgeon coders who would rather be right than to design good software, well, you truly have my pity.

Extending your pity to anyone who doesn't agree with you isn't exactly the most effective way to get your message across.



Consider Mr. T. He's been pitying fools since the early 80's, and the world is still awash in

foolishness.

It's too bad, because the message is an important one. The general adoption of unit testing is one of the most fundamental advances in software development in the last 5 to 7 years.

How do you solve a software problem? How do they teach you to handle it in school? What's the first thing you do? You think about how to solve it. You ask, "What code will I write to generate a solution?" But that's backward. The first thing you ask is not "What code will I write?" The first thing you ask is "How will I know that I've solved the problem?"

We're taught to assume we already know how to tell whether our solution works. It's a non-question. Like indecency, we'll know it when we see it. We believe we don't actually need to think, before we write our code, about what it needs to do. This belief is so deeply ingrained, it's difficult for most of us to change.

King presents a list of [12 specific ways](#) adopting a test-first mentality has helped him write better code:

1. Unit tests prove that your code actually works
2. You get a low-level regression-test suite
3. You can improve the design without breaking it
4. It's more fun to code with them than without
5. They demonstrate concrete progress
6. Unit tests are a form of sample code
7. It forces you to plan before you code
8. It reduces the cost of bugs
9. It's even better than code inspections
10. It virtually eliminates coder's block
11. Unit tests make better designs
12. It's faster than writing code without tests

Even if you only agree with a quarter of the items on that list — and I'd say at least half of

them are true in my experience — that is a huge step forward for software developers. You'll get no argument from me on the overall [importance of unit tests](#). I've increasingly come to believe that **unit tests are so important that they should be a first-class language construct**.

However, I think the test-first dogmatists tend to be a little too religious for their own good. **Asking developers to fundamentally change the way they approach writing software overnight is asking a lot**. Particularly if those developers have yet to write their first unit test. I don't think any software development shop is ready for test-first development until they've adopted unit testing as a standard methodology on every software project they undertake.

[Excessive religious fervor](#) could sour them on the entire concept of unit testing.

And that's a shame, because **any tests are better than zero tests**. And isn't unit testing just a barely more formal way of doing the ad-hoc testing we've been doing all along? I think [Fowler](#) said it best:

Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead.

I encourage developers to see the value of unit testing; I urge them to get into the habit of writing structured tests alongside their code. That small change in mindset could eventually lead to bigger shifts like test-first development — but you have to crawl before you can *sprint*.

Unit Testing versus Beta Testing

Why does Wil Shipley, the [author of Delicious Library](#), [hate unit testing so much?](#)

I've certainly known companies that do "unit testing" and other crap they've read in books. Now, you can argue this point if you'd like, because I don't have hard data; all I have is my general intuition built up over my paltry 21 years of being a professional programmer.

[..] You should test. Test and test and test. But I've NEVER, EVER seen a structured test program that (a) didn't take like 100 man-hours of setup time, (b) didn't suck down a ton of engineering resources, and (c) actually found any particularly relevant bugs. Unit testing is a great way to pay a bunch of engineers to be bored out of their minds and find not much of anything. [I know -- one of my first jobs was writing unit test code for Lighthouse Design, for the now-president of Sun Microsystems.] You'd be MUCH, MUCH better offer hiring beta testers (or, better yet, offering bug bounties to the general public).

Let me be blunt: YOU NEED TO TEST YOUR DAMN PROGRAM. Run it. Use it. Try odd things. Whack keys. Add too many items. Paste in a 2MB text file. FIND OUT HOW IT FAILS. I'M YELLING BECAUSE THIS IS IMPORTANT.

Most programmers don't know how to test their own stuff, and so when they approach testing they approach it using their programming minds: "Oh, if I just write a program to do the testing for me, it'll save me tons of time and effort."

It's hard to completely disregard the opinion of a veteran developer shipping an application that gets [excellent reviews](#). Although his opinion may seem heretical to the [Test Driven Development](#) cognoscenti, I think he has some valid points:

- **Some bugs don't matter.** Extreme unit testing may reveal.. extremely rare bugs. If a bug exists but no user ever encounters it, do you care? If a bug exists but only one in ten thousand users ever encounters it, do you care? Even Joel Spolsky [seems to agree](#) on this point. Shouldn't we be fixing bugs based on data gathered from actual usage rather than a stack of obscure, failed unit tests?

- **Real testers hate your code.** A unit test simply verifies that something works. This makes it far, far too easy on the code. Real testers hate your code and will do whatever it takes to break it — feed it garbage, send absurdly large inputs, enter unicode values, double-click every button in your app, etcetera.
- **Users are crazy.** Automated test suites are a poor substitute for real-world beta testing by actual beta testers. Users are erratic. Users have favorite code paths. Users have weird software installed on their PCs. Users are crazy, period. Machines are far too rational to test like users.

While I think basic unit testing can *complement* formal beta testing, I tend to agree with Wil: **the real and best testing occurs when you ship your software to beta testers.** If unit test coding is cutting into your beta testing schedule, you're making a very serious mistake.

Low-Fi Usability Testing

Pop quiz, hotshot. **How do you know if your application works?** Sure, maybe your app compiles. Maybe it passes all the unit tests. Maybe it ran the QA gauntlet successfully. Maybe it was successfully deployed to the production server, or packaged into an installer. Maybe your beta testers even signed off on it.

But that doesn't mean it works.

Can users actually *understand* your application? Can they *get their work done* in your application? That's what defines a working application. All the other stuff I listed is just noise. **You don't know if your application truly works until you've performed usability tests on it with actual users.**

And you regularly do usability testing on your application, right?

That's what I thought. One of the central concepts in Steve Krug's book [Don't Make Me Think](#) is that usability testing is essential to any software project. Krug calls his simplified approach to usability testing *lost our lease, going-out-of-business-sale usability testing*:

Usability testing has been around for a long time, and the basic idea is pretty simple: If you want to know whether your software or your Web site or your VCR remote control is easy enough to use, watch some people while they try to use it and note where they run into trouble. Then fix it, and test it again.

In the beginning, though, usability testing was a very expensive proposition. You had to have a usability lab with an observation room behind a one-way mirror, and at least two video cameras so you could record the users' reactions and the thing they were using. You had to recruit a lot of people so you could get results that were statistically significant. It was Science. It cost \$20,000 to \$50,000 a shot. It didn't happen very often.

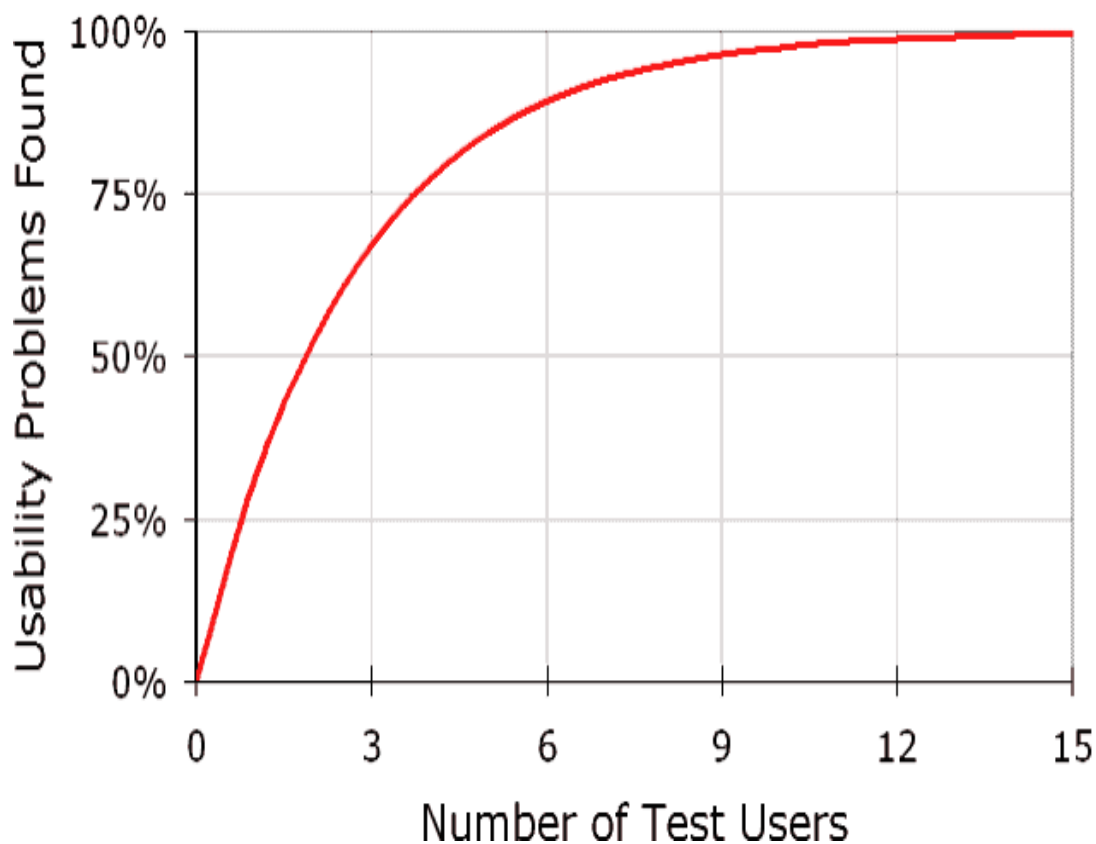
But in 1989 Jakob Nielsen wrote a paper titled ["Usability Engineering at a Discount"](#) and pointed out that it didn't have to be that way. You didn't need a usability lab, and you could achieve the same results with a lot fewer users. The idea of discount usability testing was a huge step forward. The only problem is that a decade later most people still perceive testing as a big deal, hiring someone to conduct a test still costs \$5,000 to \$15,000, and as a result it doesn't happen nearly often enough.

What I'm going to commend to you in this chapter is something even more drastic: Lost our lease, going-out-of-business-sale usability testing. I'm going to try to explain how to do your own testing when you have no money and no time. If you can afford to hire a professional to do your testing, by all means do it — but don't do it if it means you'll do less testing.

Krug points out that **usability testing is only as difficult as you make it**. It's possible to get useful results from a usability test with a *single user*, even:

[Usability] testing always works, and even the worst test with the wrong user will show you things you can do to improve your site. I make a point of always doing a live user test at my workshops so that people can see it's very easy to do and it always produces an abundance of valuable insights. I ask for a volunteer and have him try to perform a task on a site belonging to one of the other attendees. These tests last less than ten minutes, but the person whose site is being tested usually scribbles several pages of notes. And they always ask if they can have the recording of the test to show their team back home. Once person told me that after his team saw the recording, they made one change to their site which they later calculated had resulted in \$100,000 in savings.

For more proof that you don't need a lot of users to have an effective usability test, Jakob Nielsen offers [the following graph](#):



Obviously, not doing any usability testing at all is a disaster. But what's not so obvious is that usability testing with just a few users is remarkably effective. And it can be relatively painless if you follow Krug's broad guidelines for low-fidelity usability testing:

- **When should I test?** Ideally, once per month. You should be running small usability tests continuously throughout the development process. The tests should be short and simple, so you can conduct them almost any time with little advance planning.
- **How many users do I need?** Three or four, max.
- **What kind of users?** Grab some people. Anyone who can use a computer will do. The best-kept secret of usability testing is that *it doesn't much matter who you test*. It's a good idea to get representative users, but it's much more important to test early and often. Don't be embarrassed to ask friends and neighbors.
- **How much time will it take?** 45 minutes to an hour per user. Keep it simple. Keep it small. Although it does take extra time to conduct usability tests, even simple ones, ultimately you will save time. The results of the usability tests will prevent you from [wasting time arguing endlessly](#), or redoing things at the end of a project.
- **Where do I conduct the test?** Any office or conference room. All you need is a

room with a desk, a computer and two chairs where you won't be interrupted.

- **Who should do the testing?** Any reasonably patient human being. Choose someone who tends to be patient, calm, empathetic and a good listener. With a little practice, most people can get quite good at it.
- **What equipment do I need?** All you need is [some form of screen recording software](#), such as [Camtasia](#). If you want to get really fancy you can bring in a camcorder to record the person and the screen.
- **How do I prepare for the tests?** Decide what you want to show. Have [a short script](#) ready to guide the participants through the test.
- **How much will it cost?** Minus the moderator's time, a \$50-\$100 stipend per user.
- **How do we interpret the results?** Debrief the development team and any interested stakeholders over lunch the same day. One of the nicest things about usability testing is that the results tend to be obvious to everyone who's watching. The serious problems are hard to miss.

If you don't already own a copy of [Don't Make Me Think](#), shame on you. In the meantime, I highly recommend [downloading Chapter 9 of Steve Krug's Don't Make Me Think](#), which has much more detail than the summary I've presented.

Usability testing doesn't have to be complicated. If you really want to know if what you're building works, **ask someone to use it while you watch**. If nothing else, grab Joe from accounting, Sue from marketing, *grab anyone nearby who isn't directly involved with the project*, and have them try it. Don't tell them what to do. Give them a task, and remind them to think out loud while they do it. Then quietly sit back and *watch what happens*. I can tell you from personal experience that the results are often eye-opening.

The benefits of usability testing are clear. You just have to *do it* to realize any of those benefits.

What's Worse Than Crashing?

Here's an interesting thought question from Mike Stall: [what's worse than crashing?](#) Mike provides the following list of crash scenarios, in order from best to worst:

1. Application works as expected and never crashes.
2. Application crashes due to rare bugs that nobody notices or cares about.
3. Application crashes due to a commonly encountered bug.
4. Application deadlocks and stops responding due to a common bug.
5. Application crashes long after the original bug.
6. **Application causes data loss and/or corruption.**

Mike points out that there's a natural tension between...

- failing *immediately* when your program encounters a problem, eg "fail fast"
- attempting to recover from the failure state and proceed normally

The philosophy behind "fail fast" is best explained in [Jim Shore's article](#).

Some people recommend making your software robust by working around problems automatically. This results in the software "failing slowly." The program continues working right after an error but fails in strange ways later on. A system that fails fast does exactly the opposite: when a problem occurs, it fails immediately and visibly. Failing fast is a nonintuitive technique: "failing immediately and visibly" sounds like it would make your software more fragile, but it actually makes it more robust. Bugs are easier to find and fix, so fewer go into production.

Fail fast is reasonable advice — if you're a developer. What could possibly be easier than [calling everything to a screeching halt](#) the minute you get a byte of data you don't like? Computers are spectacularly unforgiving, so it's only natural for developers to reflect that masochism directly back on users.

But from the user's perspective, failing fast isn't helpful. To them, it's just another [meaningless error dialog](#) preventing them from getting their work done. The best software never pesters users with meaningless, trivial errors — it's [more considerate than that](#). Unfortunately, **attempting to help the user by fixing the error could make things worse by leading to subtle and catastrophic failures down the road**. As you work your way down Mike's list, the pain grows exponentially. For both developers *and* users. Troubleshooting #5 is a brutal death march, and by the time you get to #6 — you've lost or corrupted user data — you'll be lucky to have any users *left* to fix bugs for.

What's interesting to me is that despite causing more than my share of software crashes and hardware bluescreens, I've *never* lost data, or had my data corrupted. You'd figure Murphy's Law would force the worst possible outcome at least once a year, but it's exceedingly rare in my experience. Maybe this is an encouraging sign for the current state of software engineering. Or maybe I've just been lucky.

So what can we, as software developers, do about this? If we adopt a “fail as often and as obnoxiously as possible” strategy, we've clearly failed our users. But if we corrupt or lose our users' data through misguided attempts to prevent error messages — if we fail to treat our users' data as sacrosanct — we've *also* failed our users. You have to do both at once:

1. If you *can* safely fix the problem, you should. Take responsibility for your program. Don't slouch through the easy way out by placing the burden for dealing with every problem squarely on your users.
2. If you *can't* safely fix the problem, always err on the side of protecting the user's data. Protecting the user's data is a sacred trust. If you harm that basic contract of trust between the user and your program, you're hurting not only your credibility — but the credibility of the entire software industry as a whole. Once they've been burned by data loss or corruption, users don't soon forgive.

The guiding principle here, as always, should be to **respect your users**. Do the right thing.

Gift this book to your friends...for [free](#).

Become a Hyperink reader. Get a [special surprise](#).

Like the book? Support our author and leave a [comment](#)!

X.

Building, Managing and Benefiting from a Community



Listen To Your Community, But Don't Let Them Tell You What to Do

 **Jeff Atwood@codinghorror**

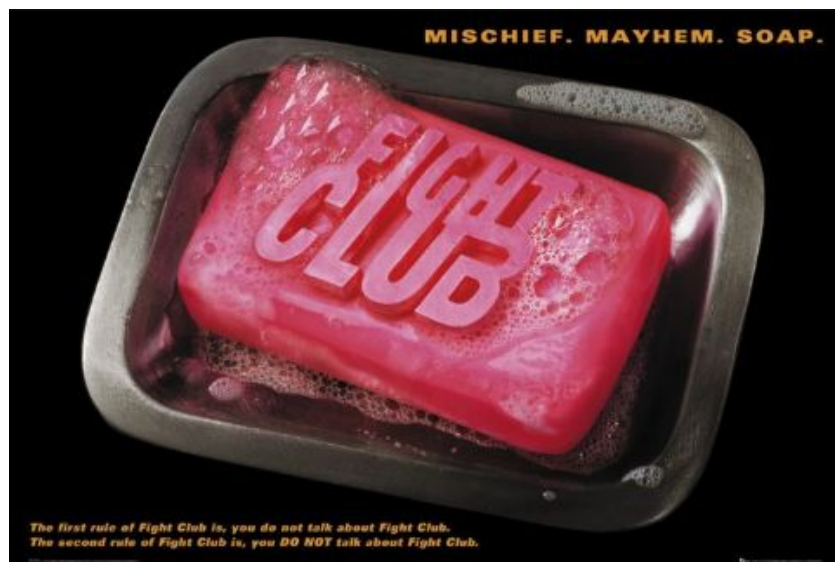
“I wish more people in our industry were trying to build communities first and products second, like photomatt with WordPress”

11:28 PM - 30 May 12

You know how interviewers love asking about your greatest weakness, or the biggest mistake you've ever made? These questions may sound formulaic, maybe even borderline cliché, but be careful when you answer: they are [more important than they seem](#).

So when people ask me **what our biggest mistake was in building Stack Overflow** I'm glad I don't have to fudge around with platitudes. I can honestly and openly point to a *huge, honking, ridiculously dumb mistake* I made from the very first day of development on Stack Overflow — and, worse, a mistake I stubbornly clung to for a solid nine-month period after that over the continued protestations of the community. I even went so far as to write a whole blog post [decrying its very existence](#).

For the longest time, I had an awfully Fight Club-esque way of looking at this: **the first rule of Stack Overflow was that you didn't discuss Stack Overflow!** After all, we were there to learn about *programming* with our peers, not learn about *a stupid website*. Right?



I didn't see the need for a meta.

Meta is, of course, the place where you go to discuss the place. Take a moment and think about what that means. Meta is for people who care so deeply about their community that they're willing to go one step further, to come together and spend even *more* of their time deciding how to maintain and govern it. So, in a nutshell, I was telling the people who *loved Stack Overflow the most of all* to basically ... f**k off and go away.

As I said, not my finest hour.

In my defense, I did eventually figure this out, thanks to the continued prodding of the community. Although we'd used an external meta site since beta, we eventually launched our very own [meta.stackoverflow](https://meta.stackoverflow.com) in June 2009, ten months after public beta. And we fixed this *very* definitively with Stack Exchange. Every [Stack Exchange site](#) we launch has a meta from day one. We now know that meta participation is the source of all meaningful leadership and governance in a community, so it is cultivated and monitored closely.

I also paid penance for my sins by becoming the top user of our own meta. I've spent the last 2 years and 7 months totally immersed in **the morass of bugs, feature requests, discussions and support that is our meta**. As you can see [in my profile](#), I've visited meta 901 unique days in that time frame, which is disturbingly close to every day. I consider my meta participation stats a badge of honor, but more than that, it's my *job* to help build this thing alongside you. We explicitly do everything in public on Stack Exchange — it's very intentionally the opposite of [Ivory Tower Development](#).

Along the way I've learned a few lessons about building software with your community,

and handling community feedback.

1. 90 percent of all community feedback is crap.

Let's get this out of the way immediately. [Sturgeon's Law](#) can't be denied by any man, woman, child ... or community, for that matter. Meta community, [I love you to death](#), so let's be honest with each other: most of the feedback and feature requests you give us are just not, uh, er ... *actionable*, for a zillion different reasons.

But take heart: **this means 10 percent of the community feedback you'll get is awesome!** I guarantee you'll find ten posts that are pure gold, that have the potential to make the site clearly better for everyone ... provided you have the intestinal fortitude to look at a hundred posts to get there. Be prepared to spend a lot of time, and I mean a *whole freaking lot of time*, mining through community feedback to extract those rare gems. I believe every community has users savvy enough to produce them in some quantity, and they're often startlingly wonderful.

2. Don't get sweet talked into building a truck.

You should immediately triage the feedback and feature requests you get into two broad buckets:

We need power windows in this car!

or

We need a truck bed in this car!

The former is, of course, a reasonable thing to request adding to a car, while the latter is a request to change the fundamental nature of the vehicle. The malleable form of software makes it all too tempting to bolt that truck bed on to our car. Why not? Users keep asking for it, and trucks sure are convenient, right?

Don't fall into this trap. Stay on mission. That car-truck hybrid is awfully tempting to a lot of folks, but then you end up with a [Subaru Brat](#). Unless you *really* want to build a truck after all, the users asking for truck features need to be gently directed to their nearest truck dealership, because they're in the wrong place.

3. Be honest about what you won't do.

It always depressed me to see bug trackers and feedback forums with thousands of items languishing there in no man's land with no status at all. That's a sign of a neglected community, and worse, a dishonest relationship with the community. It is sadly all too

typical. Don't do this!

I'm not saying you should tell your community that their feedback sucks, even when it frequently does. That'd be mean. But don't be shy about *politely* declining requests when you feel they don't make sense, or if you can't see any way they could be reasonably implemented. (You should always reserve the right to change your mind in the future, of course.) Sure, it hurts to be rejected — but it hurts far more to be *ignored*. I believe very, very strongly that if you're honest with your community, they will ultimately respect you more for that.

All relationships are predicated on honesty. If you're not willing to be honest with your community, how can you possibly expect them to respect you ... or continue the relationship?

4. Listen to your community, but don't let them tell you what to do.

It's tempting to take meta community requests as a wholesale template for development of your software or website. The point of a meta is to listen to your community, and act on that feedback, right? On the contrary, **acting too directly on community feedback is incredibly dangerous**, and the reason many of these community initiatives fail when taken too literally. I'll let Tom Preston-Werner, the co-founder of GitHub, [explain](#):

Consider a feature request such as "GitHub should let me FTP up a documentation site for my project." What this customer is really trying to say is "I want a simple way to publish content related to my project," but they're used to what's already out there, and so they pose the request in terms that are familiar to them. We could have implemented some horrible FTP based solution as requested, but we looked deeper into the underlying question and now we allow you to publish content by simply pushing a Git repository to your account. This meets requirements of both functionality and elegance.

Community feedback is great, but it should never be used as a crutch, a substitute for thinking deeply about what you're building and *why*. Always try to identify what the underlying needs are, and come up with a sensible roadmap.

5. Be there for your community.

Half of community relationships isn't doing what the community thinks they want at any given time, but **simply being there to listen and respond to the community**. When the co-founder of Stack Exchange responds to your meta post — even if it wasn't exactly what you may have wanted to hear — I hope it speaks volumes about how committed we are to really, truly building this thing alongside our community.

Regardless of whether money is changing hands or not, you should love discovering some small gem of a community request or bugfix on meta that makes your site or product better, and swooping in to make it so. That's a virtuous public feedback loop: it says *you matter* and *we care* and *everything just keeps on getting better* all in one delightful gesture.

And isn't that what it's all about?

I Repeat: Do Not Listen to Your Users

Paul Buchheit on [listening to users](#):

I wrote the first version of Gmail in one day. It was not very impressive. All I did was stuff my own email into the Google Groups (Usenet) indexing engine. I sent it out to a few people for feedback, and they said that it was somewhat useful, but it would be better if it searched over their email instead of mine. That was version two. After I released that people started wanting the ability to respond to email as well. That was version three. That process went on for a couple of years inside of Google before we released to the world.

Startups don't have hundreds of internal users, so it's important to release to the world much sooner. When [FriendFeed](#) was semi-released (private beta) in October, the product was only about two months old (and 99.9% written by two people, Bret and Jim). We've made a lot of improvements since then, and the product that we have today is much better than what we would have built had we not launched. The reason? We have users, and we listen to them, and we see which things work and which don't.

Listening to users is a tricky thing. Users often don't know what they want, and even if they did, the [communication is likely to get garbled](#) somewhere between them and you. By no means should you *ignore* your users, though. Most people will silently and forever walk away if your software or website doesn't meet their needs. The users who care enough to give you feedback deserve your attention and respect. They're essentially taking it upon themselves to design your product. If you don't listen attentively and politely respond to all customer feedback, you're setting yourself up for eventual failure.

It's rude not to listen to your users. So how do we reconcile this with the first rule of usability — [Don't Listen to Users?](#)

To discover which designs work best, watch users as they attempt to perform tasks with the user interface. This method is so simple that many people overlook it, assuming that there must be something more to usability testing. [It] boils down to the basic rules of

usability:

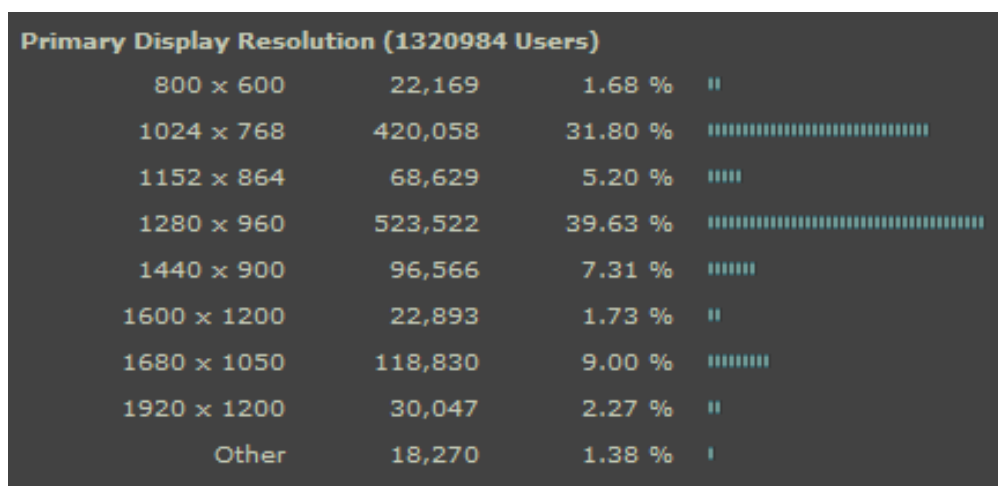
- Watch what people actually do.
- Do not believe what people say they do.
- Definitely don't believe what people predict they *may* do in the future.

I think Paul had it right, but it's easy to miss. The relevant phrase in Paul's post is **we see which things work**, which implies measurement and *correlation*. There's no need to directly watch users (although it never hurts) when you have detailed logs showing what they actually did. Collect user feedback, then correlate it with [data on what those users are actually doing](#):

Don't just implement feature requests from "user representatives" or "business analysts." The most common way to get usability wrong is to **listen to what users say rather than actually watching what they do**. Requirement specifications are always wrong. You must prototype the requirements quickly and show users something concrete to find out what they really need.

Acting on user feedback alone is questionable. No matter how well intentioned you're guessing. Why guess when you can take actions based on cold, hard data? Acting on user feedback *and* detailed usage metrics for your application or website — that's the gold standard.

Consider Valve software's [hardware survey](#). A particularly vocal set of gamers might demand support for extremely high widescreen resolutions such as 1920 x 1200 or 2560 x 1600. Understandable, since they've spent a lot of money on high-end gaming rigs. But what resolutions do most people actually play at?



Based on this survey of 1.3 million Steam users, about 10 percent of gamers have high resolution, widescreen displays. There are other reasons you might want to satisfy this request, of course. Those 10 percent tend to be the most dedicated, influential gamers. But having actual data behind your user feedback lets you vet the actions you take, to ensure that you're spending your development budget wisely. The last thing you want to do is fritter away valuable engineering time on features that almost nobody is using, and having usage data is how you tell the difference.

Valve also collects an exhaustive set of gameplay statistics for their games, such as [Team Fortress 2](#).

We've traditionally relied on things like written feedback from players to help decide which improvements to focus on. More recently, Steam has allowed us to collect more information than was previously possible. TF2 includes a reporting mechanism which tells us details about how people are playing the game. We're [sharing the data we collect](#) because we think people will find it interesting, and because we expect to spot emergent problems earlier, and ultimately build better products and experiences as a result.

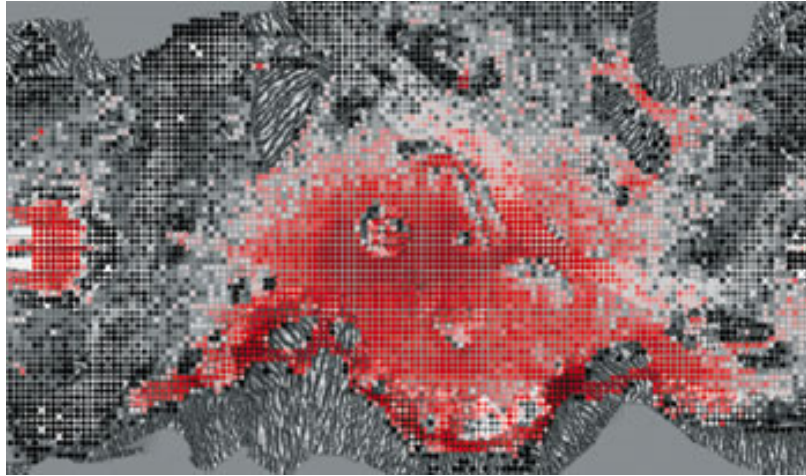
The very first graph, of **time played per class**, illustrates one problem with Team Fortress 2 in a way that I don't think any amount of player feedback ever could.

Scout	17.5%
Engineer	17.3%
Soldier	15%
Demoman	10.5%
Sniper	10.1%
Heavy	8.5%
Spy	8%
Pyro	7%
Medic	5.5%

The medic class is severely underrepresented in actual gameplay. I suppose this is because Medics don't engage in much direct combat, so they're not as exciting to play as, say, a Demoman or Soldier. That's unfortunate, because the healing abilities of the medic class are frequently critical to winning a round. So what did Valve do? They released [a giant set of medic-specific achievements](#) to encourage players to choose the Medic class more often. That's iterative game design based on actual, real-world

gameplay data.

Using detailed gameplay metrics to refine game design isn't new; [Bungie ran both Halo 2 and 3 through comprehensive usability lab tests.](#)



In April, Bungie found a nagging problem with Valhalla, one of Halo 3's multiplayer levels:

Player deaths (represented in dark red on this "heat map" of the level) were skewing toward the base on the left, indicating that forces invading from the right had a slight advantage. After reviewing this image, designers tweaked the terrain to give both armies an even chance.

Again — try to imagine how you'd figure out this fundamental map imbalance based on player feedback. I'm not sure if it's even possible.

Make sure your application or website is capturing user activity in a useful, meaningful way. User feedback is important. Don't get me wrong. But never take action *solely* based on user feedback. Always have some kind of user activity data to corroborate and support the valuable user feedback you're getting. Ignoring your user feedback may be setting yourself up for eventual failure, but blindly acting on every user request is *certain* failure.

The Gamification

When Joel Spolsky and I set out to design the [Stack Exchange](#) Q&A engine in 2008 — then known as Stack Overflow — we borrowed liberally and unapologetically from any online system that we felt worked. Some of our notable influences included:

- Reddit and Digg voting
- Xbox 360 achievements
- Wikipedia editing
- eBay karma
- Blogs and blog comments
- Classic web bulletin boards

All these elements were folded up into the Stack Exchange Q&A engine, so that we might help people create useful artifacts on the internet while learning with and among their peers. You know the old adage that *good artists copy, great artists steal*? That quote is [impossible to source](#), but it means we were *repurposing* these elements we liked.

So, what do Picasso and T.S. Eliot mean? They say, in the briefest of terms: **take old work to a new place**. Steal the Google site, strip down what works (fast load, nonexistent graphics, small quirky changes that delight) and use the parts on your own site. Look at the curve of a Coke Bottle and create a beautiful landscape painting with it. Take the hairline pinstriping on the side of somebody's car, reimagine it on your print job. Find inspiration in the world you live in, where nothing is truly new so that everything has the potential to be innovative.

Unfortunately, the elements we liked were often buried in mounds of stuff that we ... sort of hated. So extracting just the good parts and removing the rest was part of the mission. If you're lucky enough to have [a convenient villain to position yourself against](#), that might be all you need.


Traditional web bulletin board systems have a design that was apparently permanently

frozen in place circa 2001 along with Windows XP. Consider this typical forum thread.

View First Unread
Tweet
Thread Tools ▾

10-07-11, 04:51 PM
#1

Archer0915
★★★★★★
"The Expert"



Join Date: Nov 2008
Location: Inside a PCB coffin
looking for short circuits
[News Team Profile](#)

Thinking about a couple of X6 processors. Will I even see a difference? I have moved all of my encoding and gaming to Intel/nVidia for the time being until I see what the BD can actually do.

So for daily use and with a mild OC will I see a difference? I know the X6 is essentially the same as X4 as far as performance on four cores goes but will two more cores offload enough processes for me to notice.

People that buy OEM systems think Linux was a Charlie Brown character, a registry is something you see at target to buy shower gifts, RAM is a Dodge truck and a hard drive is DC at rush hour.


Current Active Fleet:

Daily Driver - 855 X4@3.6, 8GB Ram, 890GX GFX Bedroom PC - 620@3.6, 8GB Ram, 790GX GFX
 Entertainment Room HTPC & Media/File Server - G6950@3.5, 4GB Ram, On Chip GFX Living Room HTPC - 661@3.7, 10GB Ram, 6770_GFX
 Entertainment Room Cinema - E350, 4GB Ram, On Chip GFX (Projector) Game Rig - 2500K@4.5, 12GB Ram, 465GTX + 9800GT (PhysX) using intel BRT

Archer0915		Team:32		www.overclockers.com	
Team Rank: 533	Points:460,226	Team Rank: 4	Points:4,565,836,439		
Change 7d: ▼-1	24hr Avg:0	Users: 256 (7717)	24hr Avg: 5,224,494		
User Rank: 30,273	Today: 0	MUs: 9,337,830	Today: 600,885		

10-07-11, 08:54 PM
#2

I.M.O.G.
★★★★★★
Glorious Leader



Join Date: Nov 2002
Location: Rootstown, OH
[News Team Profile](#)

Depends what you do with it. In normal usage on my htpc I could tell no difference from an x4 to an x6.

I could readily tell a difference when I encode video.

[The OC Forums Way](#)

We are a team. We are a community. We are a fellowship made strong by mutual respect and shared dedication to the task of enriching all who come here.

[The OC Forums Thank You Thread](#)

ThinkpadT400|C2DT9400@2.53Ghz|4GB|60GB Vertex SSD
[http://imgg.us](#)

Put your computer to work for our OC Forum Teams!

[My Public Pictures](#)

10-07-11, 09:05 PM
#3

Archer0915
★★★★★★
"The Expert"



Join Date: Nov 2008
Location: Inside a PCB coffin
looking for short circuits
[News Team Profile](#)

Quote:

Originally Posted by **I.M.O.G.**

Depends what you do with it. In normal usage on my htpc I could tell no difference from an x4 to an x6.

I could readily tell a difference when I encode video.

As far as power usage how do they compare to an X4 because the only thing I do out of the ordinary on the two boxes I will potentially UG is BOINC. I only game on my HTPC and the frankenstein PC.

People that buy OEM systems think Linux was a Charlie Brown character, a registry is something you see at target to buy shower gifts, RAM is a Dodge truck and a hard drive is DC at rush hour.

Current Active Fleet:

Daily Driver - 855 X4@3.6, 8GB Ram, 890GX GFX Bedroom PC - 620@3.6, 8GB Ram, 790GX GFX
 Entertainment Room HTPC & Media/File Server - G6950@3.5, 4GB Ram, On Chip GFX Living Room HTPC - 661@3.7, 10GB Ram, 6770_GFX
 Entertainment Room Cinema - E350, 4GB Ram, On Chip GFX (Projector) Game Rig - 2500K@4.5, 12GB Ram, 465GTX + 9800GT (PhysX)

Here is the *actual information* from that forum thread.

Thinking about a couple of X6 processors.
Will I even see a difference? I have moved all of my encoding and gaming to Intel/nVidia for the time being until I see what the BD can actually do.

So for daily use and with a mild OC will I see a difference? I know the X6 is essentially the same as X4 as far as performance on four cores goes but will two more cores offload enough processes for me to notice.

Depends what you do with it. In normal usage on my htpc I could tell no difference from an x4 to an x6.

I could readily tell a difference when I encode video.

As far as power usage how do they compare to an X4 because the only thing I do out of the ordinary on the two boxes I will potentially UG is BOINC. I only game on my HTPC and the frankenstein PC.

Based on the original size of those screenshots, only **18 percent** of that forum thread page is content. The other 82 percent is lost to signatures, avatars, UI doohickeys and other web forum frippery that has somehow become accepted as “the way things are done.” I regularly participate in several expert niche bulletin boards of various types today, and they’re all built the same way. Nobody complains.

But they should.

This is the status quo that we’re up against. Yes, we fixed it for programmers with Stack Overflow, but why stop there? We want to liberate all the brilliant experts **stuck in these horrible Soviet-era concrete block housing forums** all over the web. We’d like to introduce them to the focused, no-nonsense [Stack Exchange Way](#), a beautiful silo of pure Q&A signal without all the associated web forum gunk.

There’s only one teeny-tiny obstacle in our way. As a great programmer I worked with once said:

It’s the damn users. They’ve ruined every program I’ve ever created.

Every web forum is the way it is *because users wanted it that way*. Yes, the design of the forum software certainly influences behavior, but the classic 2001-era web forum paradigm assumed that what users wanted made sense for the rest of the larger internet. As it turns out, [groups are their own worst enemy](#). What groups want, and what the rest of the world needs, are often two very different things. Random discussion is fine for entertainment, but it’s not particularly useful, nor does it tend to generate the kind of artifacts that will be relevant a few years from now like Wikipedia does. So then the problem becomes **how do you encourage groups to do what’s best for the world** rather than their own specific, selfish needs?

When I looked at this problem, I felt I knew the answer. But there wasn’t a word for it in 2008.

Now there is: [Gamification](#).

Gamification is the use of game design techniques and mechanics to solve problems and engage audiences. [...] Gamification works by ... taking advantage of humans' psychological predisposition to engage in gaming. The technique can encourage people to perform chores that they ordinarily consider boring, such as completing surveys, shopping, or reading web sites.

I had no idea this Wikipedia article even existed until a few months ago, but we are featured prominently in it. It is true that all our stolen ideas about reputation systems, achievements, identity and vote scoring are in place specifically to encourage the adoption of the brave new no-nonsense, all-signal Stack Exchange Q&A model. Without those incentive systems, when left to their own devices, what you get is ... well, every forum ever created. Broken by design.

Yes, [we have ulterior motives](#), but let me explain why I think gaming elements are not tacked on to the Stack Exchange Q&A engine, but a natural and essential element of the design from day one.

Learning is (supposed to be) fun

I've had this concept in my head way before the web emerged, long before anyone coined the term "Gamification" in 2010. In fact, I'd trace my inspiration for this all the way back to 1983.

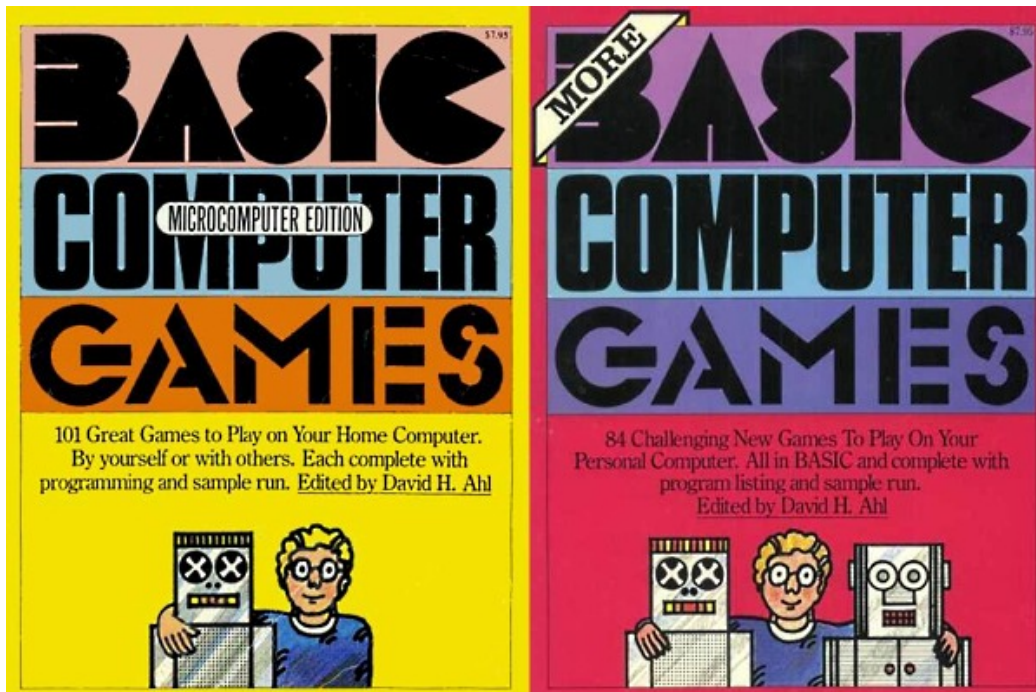


For programmers, everything we know is pretty much guaranteed to be obsolete in 10 years if we're lucky, and 5 years if we aren't. It's changing all the time. The field of

programming is almost by definition [one of constant learning](#). Programming [is supposed to be fun](#) — and it is, if you're doing it right. Nobody taught me that better than the Beagle Bros on my Apple II. Why can't learning in every *other* subject matter be just as enjoyable?

Games are learning aids

There's a long, rich history of [programmers as gamers](#). Oftentimes, the whole reason we became programmers in the first place is because we wanted to move beyond being a mere player and *change* the game, control it, modify its parameters, maybe even create our own games.



We used games to learn how to program. To a programmer, a game is [a perfectly natural introduction to real programming problems](#). I'd posit that *any* field can use games as an introduction to the subject matter — and as a reinforcement to learning.

Games help people work toward a goal

It's something of a revelation to me that solid game design can defeat the [Greater Internet F**kwad Theory](#). Two great examples of this are Counter-Strike and Team Fortress. Both games are more than ten years old, but they're still actively being played right now, by tens of thousands of people, all anonymous ... and playing as cohesive teams!

The game's objectives and rules are all cleverly constructed to make working

together the most effective way to win. None of these players know each other; the design of the game forces players to work together, whether they want to or not. It is quite literally impossible to win as a single lone wolf.



I haven't ever quite come out and said it this way, but ... I played a lot of Counter-Strike from 1998 to 2001, and **Stack Overflow is in many ways my personal Counter-Strike**. It is a programmer in Brazil learning alongside a programmer in New Jersey. Not because they're friends — but because they both love programming. The *design* of Stack Overflow makes helping your fellow programmers the most effective way to “win” and advance the craft of software development together.

And I say we all win when that happens, no matter which profession we're talking about.

I feel a little responsible for “Gamification,” since we're often cited as an example (even, much to my chagrin, on Wikipedia). I wanted to clear up exactly why we made those choices, and specifically that **all the gaming elements are there in service of a higher purpose**. I play the [Stack Exchange game](#) happily alongside everyone else, collecting reputation and badges and rank and upvotes, and I am proud to do so, because I believe it ultimately helps me become more knowledgeable and a better communicator while also improving the very fabric of the web for everyone. I hope you feel the same way.

(If you'd like to learn more about the current state of Gamification, I highly recommend [Sebastian Deterding's page](#), and specifically his [Meaningful Play: Getting Gamification Right](#) presentation.)

Suspension, Ban or Hellban?

For almost eight months after launching Stack Overflow to the public, we had no concept of banning or blocking users. Like any new frontier town in the wilderness of the internet, I suppose it was inevitable that we'd be obliged to build a jail at some point. But first we had to come up with some form of *government*.

Stack Overflow was always intended to be a democracy. With the [Stack Exchange Q&A network](#), we've come a long way towards that goal:

- We create new communities through the open, democratic process defined at [Area 51](#).
- Our communities are maintained and operated by the most avid citizens within that community. The more reputation you have, the more [privileges you earn](#).
- We hold [yearly moderator elections](#) once each community is large enough to support them.

We strive mightily to build self organizing, self-governing communities of people who are passionate about a topic, whether it be [motor vehicles](#) or [homebrewing](#) or [musical instruments](#), or ... [whatever](#). Our general philosophy is *power to the people*.



But in the absence of *some* system of law, the tiny minority of users out to do harm — intentionally or not — eventually drive out all the civil community members, leaving

behind a lawless, chaotic badland.

Our method of dealing with disruptive or destructive community members is simple: **their accounts are placed in timed suspension.** Initial suspension periods range from 1 to 7 days, and increase exponentially with each subsequent suspension. We prefer the term “timed suspension” to “ban” to emphasize that we *do* want users to come back to their accounts, *if* they can learn to refrain from engaging in those disruptive or problematic behaviors. It’s not so much a punishment as a time for the user to cool down and reflect on the nature of their participation in our community. (Well, at least in theory.)

Timed suspension works, but much like democracy itself, it is a highly imperfect, noisy system. The transparency provides ample evidence that moderators aren’t secretly whisking people away in the middle of the night. But it can also be a bit too ... *entertaining* for some members of the community, leading to hours and hours of meta-discussion about who is suspended, why they are suspended, whether it was *fair*, what the *evidence* is, how we are *censoring* people, and on and on and on. While a certain amount of introspection is important and necessary, it can also become [a substitute for getting stuff done](#). This might naturally lead one to wonder — **what if we could suspend problematic users without anyone knowing they had been suspended?**

There are three primary forms of secretly suspending users that I know of:

1. A **hellbanned** user is invisible to all other users, but crucially, not himself. From their perspective, they are participating normally in the community but *nobody ever responds to them*. They can no longer disrupt the community because they are effectively a ghost. It’s a clever way of enforcing the “don’t feed the troll” rule in the community. When nothing they post ever gets a response, a hellbanned user is likely to get bored or frustrated and leave. I believe it, too; if I learned anything from reading [The Great Brain](#) as a child, it’s that the silent treatment is the cruelest punishment of them all.

I’ve always associated hellbanning with the Something Awful Forums. Per [this amazing MetaFilter discussion](#), it turns out the roots of hellbanning go much deeper — all the way back to an early Telnet BBS system called [Citadel](#), where the “problem user bit” was introduced around 1986. Like so many other things in social software, it keeps getting reinvented over and over again by [clueless software developers](#) who believe they’re the first programmer smart enough to figure out how people work. It’s supported in most popular forum and blog software, as documented in the [Drupal Cave module](#).

(There is one additional form of hellbanning that I feel compelled to mention because it is particularly cruel – when hellbanned users can see only themselves *and other hellbanned users*. Brrr. I’m pretty sure Dante wrote a chapter about that, [somewhere](#).)

2. A **slowbanned** user has delays forcibly introduced into every page they visit. From their perspective, your site has just gotten terribly, horribly slow. And stays that way. They can hardly disrupt the community when they’re struggling to get web pages to load. There’s also science behind this one, because [per research from Google and Amazon](#), every page load delay directly reduces participation. Get slow enough, for long enough, and a slowbanned user is likely to seek out greener and speedier pastures elsewhere on the internet.
3. An **errorbanned** user has errors inserted at random into pages they visit. You might consider this a more severe extension of slowbanning — instead of pages loading slowly, they might not load at all, return cryptic HTTP errors, return the wrong page altogether, fail to load key dependencies like JavaScript and images and CSS, and so forth. I’m sure your devious little brains can imagine dozens of ways things could go “wrong” for an errorbanned user. This one is a bit more esoteric, but it isn’t theoretical; an existing implementation exists in the form of the [Drupal Misery module](#).

Because we try to hew so closely to the real-world model of democracy with Stack Exchange, I’m not quite sure how I feel about these sorts of reality-altering tricks that are impossible in the world of atoms. On some level, they feel disingenuous to me. And it’s a bit like [wishing users into the cornfield](#) with superhuman powers far beyond the ken of normal people. But I’ve also spent many painful hours trapped in public dialog about users who were, *at best*, just wasting everyone’s time. Democracy is a wonderful thing, but efficient, it ain’t.

That said, every community is different. I’ve personally talked to people in charge of large online communities — ones you probably participate in every day — and part of the reason those communities *haven’t* broken down into utter chaos by now is because they secretly **hellban** and **slowban** their most problematic users. These solutions do neatly solve the problem of getting troublesome users to “voluntarily” decide to leave a community with a minimum of drama. It’s hard to argue with techniques that are proven to work.

I think everyone has a right to know what sort of jail their community uses, even these secret, invisible ones. But keep in mind that whether it’s timed suspensions, traditional bans or exotic hellbans and beyond, the goal is the same: civil, sane, and safe online communities for everyone.

 **Jeff Atwood@codinghorror**

“If someone isn’t pissed off at your moderators at least once a day, your sites are inadequately moderated.”

12:54 AM - 24 Dec 11

Gift this book to your friends...for [free](#).

Become a Hyperink reader. Get a [special surprise](#).

Like the book? Support our author and leave a [comment](#)!

XI.

Marketing Weasels and How Not to Be One



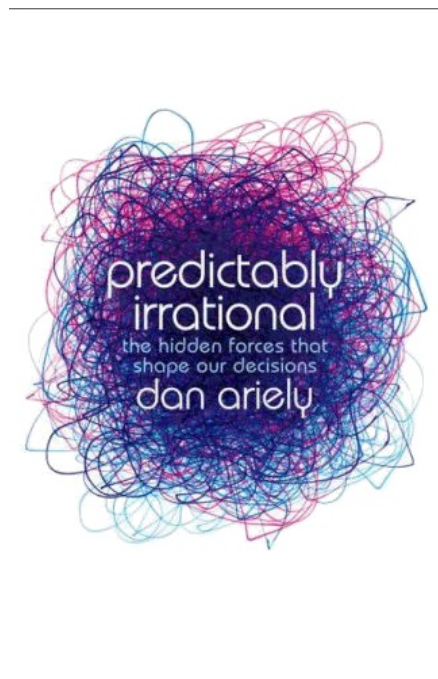
9 Ways Marketing Weasels Will Try to Manipulate You

 @codinghorror

“honestly all the Business of Software stuff absolutely bores me to tears. I just want to build awesome things that matter.”

11:17 AM - 20 Apr 12

I recently read [Predictably Irrational](#).



It's a fascinating examination of why human beings are wired and conditioned to react irrationally. We human beings are a selfish bunch, so it's all the more surprising to see how easily we can be manipulated to behave in ways that run counter to our own self-interest.

This isn't just a "gee-whiz" observation; understanding how and why we behave irrationally is important. If you don't understand how these irrational behaviors are triggered, **the marketing weasels will use them against you.**

In fact, it's already happening. Witness [10 Irrational Human Behaviors and How to Leverage Them to Improve Web Marketing](#). Don't say I didn't warn you.

Let's take a look at the various excerpts presented in that article, and consider **how we can avoid falling into the rut of predictably irrational behavior** — and defend ourselves from those vicious marketing weasels.

1. Encourage false comparisons

When Williams-Sonoma introduced bread machines, sales were slow. When they added a “deluxe” version that was 50 percent more expensive, they started flying off the shelves; the first bread machine now appeared to be a bargain.

When contemplating the purchase of a \$25 pen, the majority of subjects would drive to another store 15 minutes away to save \$7. When contemplating the purchase of a \$455 suit, the majority of subjects would not drive to another store 15 minutes away to save \$7. The amount saved and time involved are the same, but people make very different choices. Watch out for relative thinking; it comes naturally to all of us.

- Realize that some premium options exist as decoys — that is, they are there only to make the less expensive options look more appealing, because they're easy to compare. Don't make binding decisions solely based on how easy it is to compare two side-by-side options from the same vendor. Try comparing all the alternatives, even those from other vendors.
- Don't be swayed by relative percentages for small dollar amounts. Yes, you saved 25 percent, but how much effort and time did you expend on that seven bucks?

2. Reinforce Anchoring

Savador Assael, the Pearl King, single-handedly created the market for black pearls, which were unknown in the industry before 1973. His first attempt to market the pearls was an utter failure; he didn't sell a single pearl. So he went to his friend, Harry Winston, and had Winston put them in the window of his 5th Avenue store with an outrageous price tag attached. Then he ran full page ads in glossy magazines with black pearls next to diamonds, rubies and emeralds. Soon, black pearls were considered precious.

Simonsohn and Loewenstein found that people who move to a new city remain anchored to the prices they paid in their previous city. People who move from Lubbock to Pittsburgh squeeze their families into smaller houses to pay the same amount. People who move from LA to Pittsburgh don't save money, they just move into mansions.

- Scale your purchases to your needs, not your circumstances or wallet size. What do you actually use? How much do you use it, and how frequently?
- Try to objectively measure the value of what you're buying; don't be tricked into measuring relative to similar products or competitors. How much does buying this save you or your company? How much benefit will you get out of it? Attempt to measure that benefit by putting a concrete dollar amount on it.

3. It's "Free"!

Ariely, Shampanier, and Mazar conducted an experiment using Lindt truffles and Hershey's Kisses. When a truffle was \$0.15 and a kiss was \$0.01, 73 percent of subjects chose the truffle and 27 percent the Kiss. But when a truffle was \$0.14 and a kiss was *free*, 69 percent chose the kiss and 31 percent the truffle.

According to standard economic theory, the price reduction shouldn't have led to any behavior change, but it did.

Ariely's theory is that for normal transactions, we consider both upside and downside. But when something is free, we forget about the downside. "Free" makes us perceive what is being offered as immensely more valuable than it really is. Humans are loss-averse; when considering a normal purchase, loss-aversion comes into play. But when an item is free, there is no visible possibility of loss.

- You will tend to overestimate the value of items you get for free. Resist this by viewing free stuff skeptically rather than welcoming it with open arms. If it was really that great, why would it be free?
- Free stuff often comes with well hidden and subtle strings attached. How will using a free service or obtaining a free item influence your future choices? What paid alternatives are you avoiding by choosing the free route, and why?
- How much effort will the free option cost you? Are there non-free options which would cost less in time or effort? How much is your time worth?
- When you use a free service or product, you are implicitly endorsing and encouraging the provider, effectively beating a path to their door. Is this something you are comfortable with?

4. Exploit social norms

The AARP asked lawyers to participate in a program where they would offer their services to needy employees for a discounted price of \$30/hour. No dice. When the program manager instead asked if they'd offer their services for free, the lawyers overwhelmingly said they would participate.

- Companies may appeal to your innate sense of community or public good to convince you to do their work at zero pay. Consider carefully before choosing to participate; what do *you* get out of contributing your time and effort? Is this truly a worthy cause? Would this be worth doing if it was a paid gig?
- When it comes to the web, make sure you aren't being turned into [a digital sharecropper](#).

5. Design for Procrastination

Ariely conducted an experiment on his class. Students were required to write three papers. Ariely asked the first group to commit to dates by which they would turn in each paper. Late papers would be penalized 1 percent per day. There was no penalty for turning papers in early. The logical response is to commit to turning all three papers in on the last day of class. The second group was given no deadlines; all three papers were due in the last day of class. The third group was directed to turn their papers in on the 4th, 8th and 12th weeks.

The results? Group 3 (imposed deadlines) got the best grades. Group 2 (no deadlines) got the worst grades, and Group 1 (self-selected deadlines) finished in the middle. Allowing students to pre-commit to deadlines improved performance. Students who spaced out their commitments did well; students who did the logical thing and gave no commitments did badly.

- Steer clear of offers of low-rate trial periods which auto-convert into automatic recurring monthly billing. They know that most people will procrastinate and forget to cancel before the recurring billing kicks in.
- Either favor fixed-rate, fixed-term plans — or become meticulous about cancelling recurring services when you're not using them.

6. Utilize the Endowment Effect

Ariely and Carmon conducted an experiment on Duke students, who sleep out for weeks to get basketball tickets; even those who sleep out are still subjected to a lottery at the end. Some students get tickets, some don't. The students who didn't get tickets told

Ariely that they'd be willing to pay up to \$170 for tickets. The students who did get the tickets told Ariely that they wouldn't accept less than \$2,400 for their tickets.

There are three fundamental quirks of human nature. We fall in love with what we already have. We focus on what we might lose, rather than what we might gain. We assume that other people will see the transaction from the same perspective as we do.

- The value of what you've spent so far on a service, product, or relationship — in effort or money — is probably far less than you think. Be willing to walk away.
- Once you've bought something, never rely on your internal judgment to assess its value, because you're too close to it now. Ask other people what they'd pay for this service, product, or relationship. Objectively research what others pay online.

7. Capitalize on our Aversion to Loss

Ariely and Shin conducted an experiment on MIT students. They devised a computer game which offered players three doors: Red, Blue and Green. You started with 100 clicks. You clicked to enter a room. Once in a room, each click netted you between 1-10 cents. You could also switch rooms (at the cost of a click). The rooms were programmed to provide different levels of rewards (there was variation within each room's payoffs, but it was pretty easy to tell which one provided the best payout).

Players tended to try all three rooms, figure out which one had the highest payout, and then spend all their time there. (These are MIT students we're talking about). Then, however, Ariely introduced a new wrinkle: Any door left unvisited for 12 clicks would disappear forever. With each click, the unclicked doors shrank by 1/12th.

Now, players jumped from door to door, trying to keep their options open. They made 15 percent less money; in fact, by choosing any of the doors and sticking with it, they could have made more money.

Ariely increased the cost of opening a door to 3 cents; no change — players still seemed compelled to keeping their options open. Ariely told participants the exact monetary payoff of each door; no change. Ariely allowed participants as many practice runs as they wanted before the actual experiment; no change. Ariely changed the rules so that any door could be "reincarnated" with a single click; no change.

Players just couldn't tolerate the idea of the loss, and so they did whatever was necessary to prevent their doors from closing, even though disappearance had no real consequences and could be easily reversed. We feel compelled to preserve options, even

at great expense, even when it doesn't make sense.

- If your choices are artificially narrowed, don't passively get funneled toward the goal they're herding you toward. Demand choice, even if it means switching vendors or allegiances.
- Don't pay extra for options, unless you can point to hard evidence that you *need* those options. Some options exist just to make you doubt yourself, so you'll worry about not having them.

8. Engender Unreasonable Expectations

Ariely, Lee, and Frederick conducted yet another experiment on MIT students. They let students taste two different beers, and then choose to get a free pint of one of the brews. Brew A was Budweiser. Brew B was Budweiser, plus 2 drops of balsamic vinegar per ounce.

When students were not told about the nature of the beers, they overwhelmingly chose the balsamic beer. When students were told about the true nature of the beers, they overwhelmingly chose the Budweiser. If you tell people up front that something might be distasteful, the odds are good they'll end up agreeing with you — because of their expectations.

- Whatever you've heard about a brand, company, or product — there's no substitute for your own hands-on experience. Let your own opinions guide you, not the opinions of others.
- Just because something is labelled "premium" or "pro" or "award-winning" doesn't mean it is. Research these claims; don't let marketing set your expectations. Rely on evidence and facts.

9. Leverage Pricing Bias

Ariely, Waber, Shiv and Carmon made up a fake painkiller, Veladone-Rx. An attractive woman in a business suit (with a faint Russian accent) told subjects that 92 percent of patients receiving VR reported significant pain relief in 10 minutes, with relief lasting up to 8 hours.

When told that the drug cost \$2.50 per dose, nearly all of the subjects reported pain relief. When told that the drug cost \$0.10 per dose, only half of the subjects reported pain relief. The more pain a person experienced, the more pronounced the effect. A similar study at U Iowa showed that students who paid list price for cold medications reported

better medical outcomes than those who bought discount (but clinically identical) drugs.

- Price often has nothing to do with value. Expensive is not synonymous with quality. Investigate whether the price is justified; never accept it at face value.
- Don't fall prey to the "moneymoon." Just because you paid for something doesn't mean it's automatically worthwhile. Not everything we pay money for works well, or was even worth what we spent for it. We all make mistakes when buying things, but we don't want to admit it.

What I learned from [Predictably Irrational](#) is that **everyone is irrational sometimes, and that's OK**. We're not perfectly logical Vulcans, after all. The trick is training yourself to know when you're *most likely* to make irrational choices, and to resist those impulses.

If you aren't at least aware of our sad, irrational human condition, well ... that's exactly where the marketing weasels want you.

How Not to Advertise on the Internet

Games that run in your web browser are all the rage, and [understandably so](#). Why not build your game for the largest audience in the world, using freely available technology, and pay zero licensing fees? One such game is Evony, formerly known as Civony — a browser-based clone of [the game Civilization](#) with [a buy-in mechanism](#).

There are also plentiful opportunities to ‘pay money’ now. In the end, Civony is still a business. And to be honest, it’s probably better to give the option for some elite folks to finance the game for the masses than to make everyone pay a subscription or watch in-game ads. In addition to the old \$0.30 per line world chat, you can spend money to speed up resource gathering, boost stats and buy in-game artifacts. I’m sure there are other ways to pay money that I haven’t discovered yet. But whenever you see a green plus-sign (+), you know the option exists to pay money for a perk.

The game is ostensibly free, but supported by a tiny fraction of players making cash payments for optional items (sometimes referred to as [“freemium”](#)). Thus, the player base needs to be quite large for the business of running the game to be sustainable, and the game’s creators **regularly purchase internet ad space to promote their game**. The most interesting thing about Evony isn’t the game, per se, but the game’s advertising. Here’s one of the early ads.



Totally reasonable advertisement. Gets the idea across that this is some sort of game [set in medieval times](#), and emphasizes the free angle.

Apparently that ad didn't perform up to expectations at Evony world HQ, because the ads got progressively ... well, take a look for yourself. These are presented in **chronological order of appearance on the internet**.



(if this lady looks familiar, [there's a reason](#).)





To be clear, **these are real ads that were served on the internet.** This is *not* a parody. Just to prove it, here's a screenshot of the last ad in context at [The Elder Scrolls Nexus](#).

The screenshot shows a web browser window with the URL <http://www.tesnexus.com/>. The website header includes a navigation menu with links for HOME, FILES, FORUMS, IMAGE SHARE, SEARCH, LOGIN, and REGISTER. The main content area features a blog post titled "Webmaster's mini blog" dated 19:14, 18 June 2009, with 55 comments. The post is titled "When the bubble bursts" and discusses the author's experience with exams and handwriting. To the right of the text is a photograph of a young man smiling. Below the text is a paragraph starting with "So what now?". An advertisement for "PLAY" is visible on the right side of the page, featuring a woman in a black lace bra and the text "Best Free Web Game. PLAY EVERY FREE FOREVER. PLAY NOW SECRETLY". At the bottom right, there is a "Site" menu with links for News, News Archive, About Us, and Site history, and a "Latest news" section with links for Rating system overhaul, Down-time over, and Scheduled site down-time today.

I've talked about [advertising responsibly](#) in the past. This is about as far in the opposite direction as I could possibly imagine. It's yet another way, sadly, [the brilliant satire Idiocracy](#) turned out to be right on the nose.



The dystopian future of [Idiocracy](#) predicted the reduction of advertising to the inevitable lowest common denominator of all, with Starbucks Exotic Coffee for Men, H.R. Block “Adult” Tax Return (home of the gentleman’s rebate), and El Pollo Loco chicken advertising a Bucket of Wings with “full release.”

Evony, thanks for showing us what it means to **take advertising on the internet to the absolute rock bottom** ... then dig a sub-basement under that, and keep on digging until you reach the white-hot molten core of the Earth. I’ve always wondered what that would be like. I guess now I know.

 **Jeff Atwood@codinghorror**

“man, Twitter has really ramped up the advertising. BROUGHT TO YOU BY CARL’S JR”

11:05 AM - 31 May 12

Groundhog Day, or, the Problem With A/B Testing

On a recent airplane flight, I happened to catch the movie [Groundhog Day](#). Again.



If you aren't familiar with this classic film, the premise is simple: Bill Murray, somehow, gets stuck reliving the same day over and over.

It's been at least 5 years since I've seen Groundhog Day. I don't know if it's my advanced age, or what, but it really struck me on this particular viewing: this is no comedy. There's a veneer of broad comedy, yes, but **lurking just under that veneer is a deep, dark existential conundrum.**

It might be amusing to relive the same day a few times, maybe even a few dozen times.

But an entire year of the same day — an entire *decade* of the same day — everything happening in precisely, exactly the same way? My back of the envelope calculation easily ran to a decade. But I was wrong. The director, Harold Ramis [thinks it was actually 30 or 40 years](#).

I think the 10-year estimate is too short. It takes at least 10 years to get good at anything, and allotting for the down time and misguided years [Phil] spent, it had to be more like 30 or 40 years [spent reliving the same day].

We only see bits and pieces of the full experience in the movie, but this time my mind began filling in the gaps. Repeating the same day for *decades* plays to our secret collective fear that our lives are irrelevant and ultimately pointless. None of our actions — even suicide, in endless grisly permutations — ever change anything. What's the point? Why bother? How many of us are trapped in here, and how can we escape?

This is some dark, scary stuff when you really think about it.

You want a prediction about the weather, you're asking the wrong Phil.

I'll give you a winter prediction.

It's gonna be cold,

it's gonna be gray,

and it's gonna last you for the rest of your life.

Comedy, my ass. I wanted to cry.

But there is a way out: redemption through repetition. If you have to watch Groundhog Day a few times to appreciate it, you're not alone. Indeed, that seems to be the whole point. Just [ask Roger Ebert](#):

“Groundhog Day” is a film that finds its note and purpose so precisely that its genius may not be immediately noticeable. It unfolds so inevitably, is so entertaining, so apparently effortless, that you have to stand back and slap yourself before you see how good it really is.

Certainly I underrated it in my original review; I enjoyed it so easily that I was seduced into cheerful moderation. But there are a few films, and this is one of them, that burrow into our memories and become reference points. When you find yourself needing the phrase This is like “Groundhog Day” to explain how you feel, a movie has accomplished something.

There's something delightfully [Ouroboros](#) about the epiphanies and layered revelations in repeated viewings of a movie *that is itself about (nearly) endless repetition*.

Which, naturally, **brings me to A/B testing**. That's what Phil spends most of those thirty years doing. He spends it pursuing a woman, technically, but it's *how* he does it that is interesting:

Rita: This whole day has just been one long setup.

Phil: It hasn't.

Rita: And I hate fudge!

Phil: [making a mental list] No white chocolate. No fudge.

Rita: What are you doing? Are you making some kind of list? Did you call my friends and ask what I like and what I don't like? Is this what love is for you?

Phil: This is real. This is love.

Rita: Stop saying that! You must be crazy.

Phil doesn't just go on one date with Rita, he goes on *thousands* of dates. During each date, he makes note of what she likes and responds to, and drops everything she doesn't. At the end he arrives at — quite literally — the perfect date. Everything that happens is the most ideal, most desirable version of all possible outcomes on that date on that particular day. Such are the luxuries afforded to a man repeating the same day forever.



This is the purest form of A/B testing imaginable. Given two choices, pick the one that “wins”, and keep repeating this ad infinitum until you arrive at the ultimate, most scientifically desirable choice. Your [marketing weasels](#) would probably collapse in an

ecstatic, religious fervor if they could achieve anything even remotely close to the level of perfect A/B testing depicted in Groundhog Day.

But at the end of this perfect date, something impossible happens: **Rita rejects Phil.**

Phil wasn't making these choices because he honestly believed in them. He was making these choices because he wanted a specific outcome — winning over Rita — and the experimental data told him which path he should take. Although the date was technically perfect, it didn't ring true to Rita, and that made all the difference.

That's the problem with A/B testing. It's empty. It has no feeling, no empathy, and at worst, [it's dishonest](#). As my friend Nathan Bowers [said](#):

A/B testing is like sandpaper. You can use it to smooth out details, but you can't actually create anything with it.

The next time you reach for A/B testing tools, remember what happened to Phil. You can achieve a shallow local maximum with A/B testing — but you'll never win hearts and minds. If you, or anyone on your team, is still having trouble figuring that out, well, the solution is simple.

Just watch Groundhog Day again.

If it Looks Corporate, Change It

Are you familiar with [happy talk](#)?

If you're not sure whether something is happy talk, there's one sure-fire test: if you listen very closely while you're reading it, you can actually hear a tiny voice in the back of your head saying "Blah blah blah blah blah...."

A lot of happy talk is the kind of self-congratulatory promotional writing that you find in badly written brochures. Unlike good promotional copy, it conveys no useful information, and focuses on saying how great we are, as opposed to delineating what makes us great.

Happy talk is the kudzu of the internet; the place is lousy with the stuff.

And then there's the visual equivalent of happy talk. Those cloying, meaningless stock photos of happy users doing ... *something* ... with a computer.



What is going on here? Given the beatific expressions, you'd think they were undergoing some kind of nerd rapture. Maybe they're getting a sneak preview of [the singularity](#), I don't know.

It's unclear to me why companies (and even some individuals) think they need happy talk, stock photos of multicultural computer users, or the occasional [headset hottie](#). Jason Cohen [provides an explanation](#):

Even before I had a single customer, I "knew" it was important to look professional. My website would need to look and feel like a "real company." I need culture-neutral language complimenting culturally-diverse clip-art photos of frighteningly chipper co-workers huddled around a laptop, awash with the thrill and delight of configuring a JDBC connection to SQL Server 2008.

It also means adopting typical "marketing-speak," so my "About Us" page started with:

Smart Bear is the leading provider of enterprise version control data-mining tools. Companies world-wide use Smart Bear's Code Historian software for risk-analysis, root-cause discovery, and software development decision-support.

“Leading provider?” “Data mining?” I’m not even sure what that means. But you have to give me credit for an impressive quantity of hyphens.

That’s what you’re supposed to do, right? That’s what other companies do, so it must be right. Who am I to break with tradition?

I’m not sure where we got our ideas about this stuff, but it is true that some large companies promote a kind of doublespeak “professionalism.” [Kathy Sierra describes her experiences at Sun:](#)

By the time I got to Sun, using the word “cool” in a customer training document was enough to warrant an entry in your annual performance eval. And not in a good way.

I cannot count the times I heard the word “professionalism” used as justification for why we couldn’t do something. But I can count the few times I heard the word “passion” used in a meeting where the goal was to get developers to adopt our newest Java technologies. What changed?

Some argue that by maintaining strict professionalism, we can get the more conservative, professional clients and thus grow the business. Is this true? Do we really need these clients? Isn’t it possible that we might even grow more if we became braver?

It’s a shame that this misguided sense of professionalism is sometimes used as an excuse to put up weird, Orwellian communication barriers between yourself and the world. At best it is a facade to hide behind; at worst it encourages us to emulate so much of what is wrong with large companies. Allow me to paraphrase the [simple advice of Elmore Leonard:](#)

If it looks corporate, change it.

The next time you find yourself using *professional* text, or *professional* stock images, consider the value of this “professionalism.” Is it legitimately helping you communicate? Or is it getting in the way?

Software Pricing: Are We Doing it Wrong?

One of the side effects of [using the iPhone App store so much](#) is that it's started to fundamentally alter my perception of software pricing. So many excellent iPhone applications are either free, or no more than a few bucks at most. That's below the threshold of impulse purchase and squarely in no-brainer territory for anything decent that I happen to be interested in.

But applications that cost \$5 or more? *Outrageous! Highway robbery!*

This is all very strange, as a guy who is used to spending at least \$30 for software of any consequence whatsoever. I love [supporting my fellow software developers with my wallet](#), and the iPhone App Store has never made that easier.

While there's an [odd aspect of race to the bottom](#) that I'm not sure is entirely healthy for the iPhone app ecosystem, the idea that **software should be priced low enough to pass the average user's "why not" threshold** is a powerful one.

What I think isn't well understood here is that low prices can be a force multiplier all out of proportion to the absolute reduction in price. Valve software has been aggressively experimenting in this area; consider the [example of the game Left 4 Dead](#):

Valve co-founder Gabe Newell announced during a DICE keynote today that last weekend's half-price sale of Left 4 Dead resulted in a 3000 percent increase in sales of the game, posting overall sales (in dollar amount) that beat the title's original launch performance.

It's sobering to think that cutting the price in half, months later, made *more* money for Valve in total than launching the game at its original \$49.95 price point. (And, incidentally, that's the price I paid for it. No worries, I got my fifty bucks worth of gameplay out of this excellent game months ago.)

The experiments didn't end there. Observe the utterly non-linear scale at work as the price of software is experimentally reduced even further on their [Steam](#) network:

The massive [Steam](#) holiday sale was also a big win for Valve and its partners. The following holiday sales data was released, showing the sales breakdown organized by price reduction:

- 10% sale = 35% increase in sales (real dollars, not units shipped)
- 25% sale = 245% increase in sales
- 50% sale = 320% increase in sales
- 75% sale = 1470% increase in sales

Note that *these are total dollar sale amounts!* Let's use some fake numbers to illustrate how dramatic the difference really is. Let's say our hypothetical game costs \$40, and we sold **100 copies** of it at that price.

Original price	Discount	Sale Price	Total Sales
\$40	none	\$40	\$4,000
\$40	10%	\$36	\$5,400
\$40	25%	\$30	\$9,800
\$40	50%	\$20	\$12,800
\$40	75%	\$10	\$58,800

If this pattern Valve documented holds true, and if my experience on the iPhone App store is any indication, **we've been doing software pricing completely wrong**. At least [for digitally distributed software](#), anyway.

In particular, I've always felt that Microsoft has priced their operating system upgrades far, far too high — and would have sold a *ton* more licenses if they had sold them at the “heck, why not?” level. For example, take a look at these upgrade options:

Mac OS X 10.6 Upgrade	\$29
Microsoft Windows 7 Home Premium Upgrade	\$119

Putting aside [schoolyard OS rivalries](#) for a moment, which one of these would you be more likely to buy? I realize this isn't entirely a fair comparison, so if \$29 seems as bonkers to you as an application for 99 cents — which I'd argue is much less crazy than it sounds — then fine. Say the Windows 7 upgrade price was a more rational \$49, or \$69. I'm sure the thought of that drives the Redmond [consumer surplus capturing marketing weasels](#) apoplectic. But the Valve data — and my own gut intuition — leads me to believe

that they'd actually make *more* money if they priced their software at the "why not?" level.

I'm not saying these pricing rules should apply to *every* market and *every* type of software in the world. But for software sold in high volumes to a large audience, I believe they might. At the very least, if you sell software, you might consider experimenting with pricing, as Valve has. You could be pleasantly surprised.

[I love buying software](#), and I know I buy a *heck* of a lot more of it when it's priced right. So why not?

Gift this book to your friends...for [free](#).

Become a Hyperink reader. Get a [special surprise](#).

Like the book? Support our author and leave a [comment](#)!

XII.

Keeping Your Priorities Straight



Buying Happiness

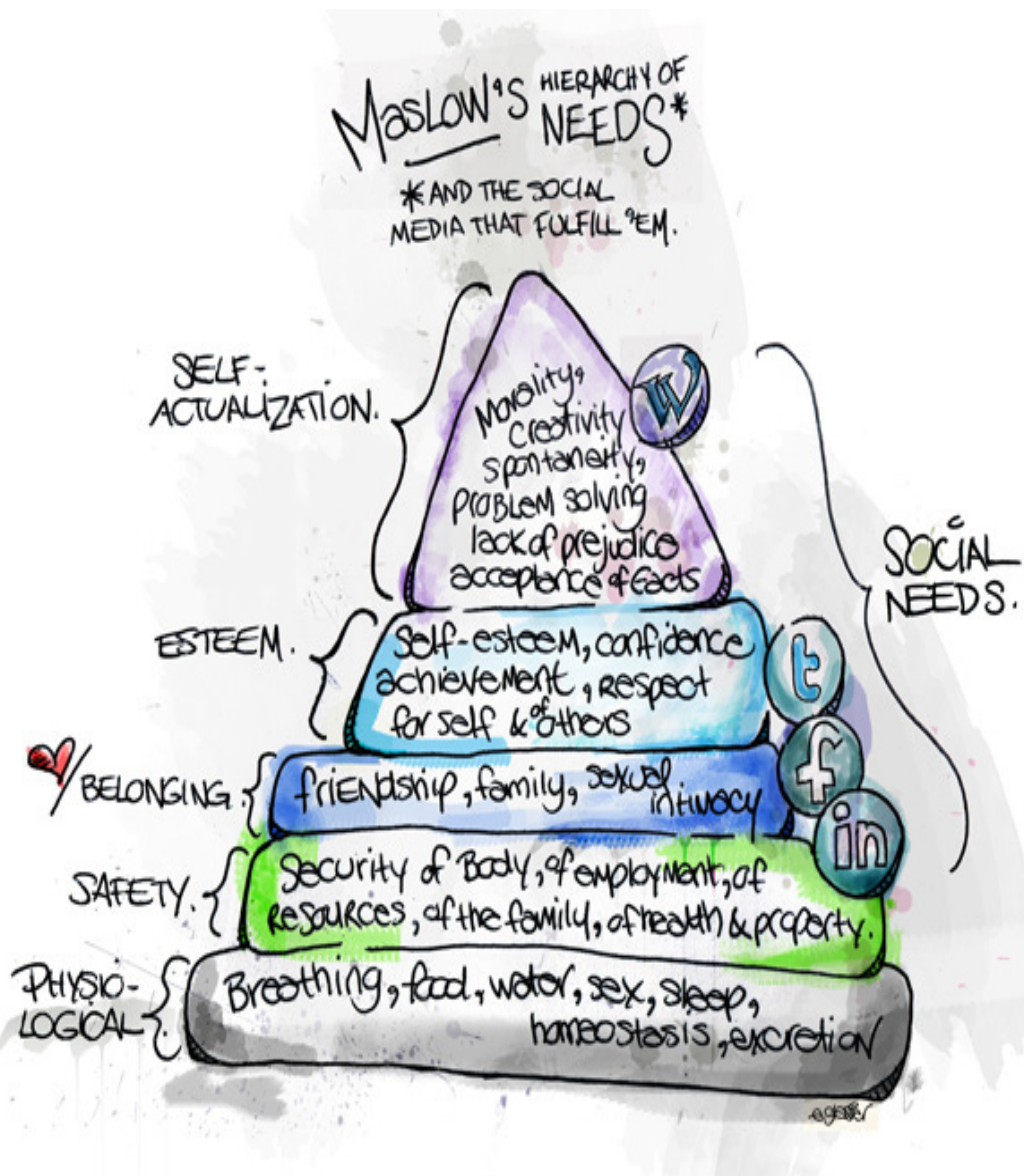
“...the hard part is figuring out why you are working all those long hours.”

Despite popular assertions to the contrary, science tells us that [money can buy happiness](#). To a point.

Recent research has begun to distinguish two aspects of subjective well-being. Emotional well-being refers to the emotional quality of an individual’s everyday experience — the frequency and intensity of experiences of joy, stress, sadness, anger and affection that make one’s life pleasant or unpleasant. Life evaluation refers to the thoughts that people have about their life when they think about it. We raise the question of whether money buys happiness, separately for these two aspects of well-being. We report an analysis of more than 450,000 responses to the Gallup-Healthways Well-Being Index, a daily survey of 1,000 US residents conducted by the Gallup Organization. [...] When plotted against log income, life evaluation rises steadily. **Emotional well-being also rises with log income, but there is no further progress beyond an annual income of \$75,000.**

For reference, the [federal poverty level](#) for a family of four is currently **\$23,050**. Once you reach a little over 3 times the poverty level in income, you’ve achieved peak happiness, as least far as money alone can reasonably get you.

This is something I’ve seen echoed in a number of studies. Once you have “enough” money to satisfy the basic items at the foot of the [Maslow’s Hierarchy of Needs](#) pyramid — that is, you no longer have to worry about food, shelter, security and perhaps having a bit of extra discretionary money for the unknown — stacking even more money up doesn’t do much, if anything, to help you scale the top of the pyramid.



But even if you're fortunate enough to have a good income, how you spend your money has a strong influence on how happy — or unhappy — it will make you. And, again, there's science behind this. The relevant research is summarized in [If money doesn't make you happy, then you probably aren't spending it right.](#)

Most people don't know the basic scientific facts about happiness — about what brings it and what sustains it — and so they don't know how to use their money to acquire it. It is not surprising when wealthy people who know nothing about wine end up with cellars that aren't that much better stocked than their neighbors', and it should not be surprising when wealthy people who know nothing about happiness end up with lives that aren't that much happier than anyone else's. Money is an opportunity for happiness, but it is an opportunity that people routinely squander because the things they think will make them happy often don't.

You may also recognize some of the authors on this paper, in particular Dan Gilbert, who also wrote the excellent book [Stumbling on Happiness](#) that touched on many of the same themes.

What is, then, the **science of happiness**? I'll summarize the basic eight points as best I can, but [read the actual paper](#) to obtain the citations and details on the underlying studies underpinning each of these principles.

1. Buy experiences instead of things

Things get old. Things become ordinary. Things stay the same. Things wear out. Things are difficult to share. But experiences are totally unique; they shine like diamonds in your memory, often more brightly every year, and they can be shared forever. Whenever possible, spend money on *experiences* such as taking your family to Disney World, rather than *things* like a new television.

2. Help others instead of yourself

Human beings are intensely social animals. Anything we can do with money to create deeper connections with other human beings tends to tighten our social connections and reinforce positive feelings about ourselves and others. Imagine ways you can spend some part of your money to help others – even in a very small way – and integrate that into your regular spending habits.

3. Buy many small pleasures instead of few big ones

Because we adapt so readily to change, the most effective use of your money is to bring frequent change, not just “big bang” changes that you will quickly grow acclimated to. Break up large purchases, when possible, into smaller ones over time so that you can savor the entire experience. When it comes to happiness, frequency is more important than intensity. Embrace the idea that lots of small, pleasurable purchases are actually *more* effective than a single giant one.

4. Buy less insurance

Humans adapt readily to both positive and *negative* change. Extended warranties and insurance prey on your impulse for loss aversion, but because we are so adaptable, people experience far less regret than they anticipate when their purchases don't work out.

Furthermore, having the easy “out” of insurance or a generous return policy can paradoxically lead to even *more* angst and unhappiness because people deprived

themselves of the emotional benefit of full commitment. Thus, avoid buying insurance, and don't seek out generous return policies.

5. Pay now and consume later

Immediate gratification can lead you to make purchases you can't afford, or may not even truly want. Impulse buying also deprives you of the distance necessary to make reasoned decisions. It eliminates any sense of anticipation, which is a strong source of happiness. For maximum happiness, savor (maybe even prolong!) the uncertainty of deciding whether to buy, what to buy, and the time waiting for the object of your desire to arrive.

6. Think about what you're not thinking about

We tend to gloss over details when considering future purchases, but research shows that our happiness (or unhappiness) largely lies in exactly those tiny details we aren't thinking about. Before making a major purchase, consider the mechanics and logistics of owning this thing, and where your actual time will be spent once you own it. Try to imagine a typical day in your life, in some detail, hour by hour: how will it be affected by this purchase?

7. Beware of comparison shopping

Comparison shopping focuses us on attributes of products that arbitrarily distinguish one product from another, but have nothing to do with how much we'll *enjoy* the purchase. They emphasize characteristics we care about while shopping, but not necessarily what we'll care about when actually using or consuming what we just bought. In other words, getting a great deal on cheap chocolate for \$2 may not matter if it's not pleasurable to eat. Don't get tricked into comparing for the sake of comparison; try to weight only those criteria that actually matter to your enjoyment or the experience.

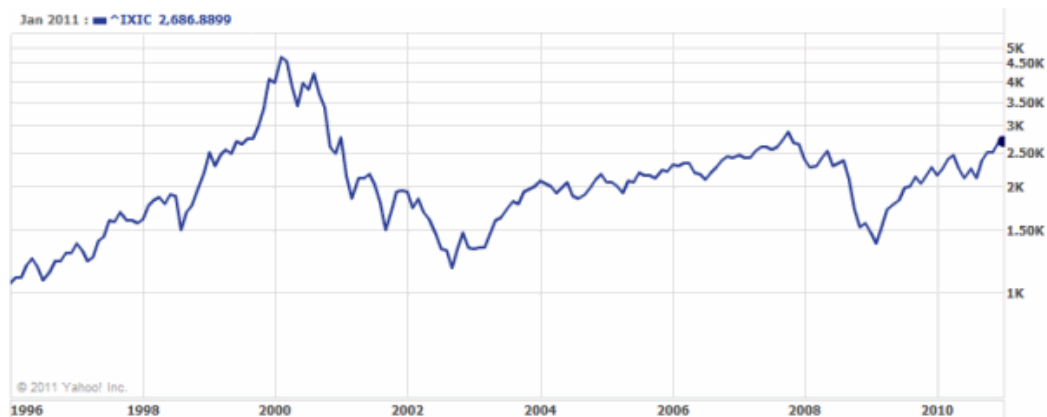
8. Follow the herd instead of your head

Don't overestimate your ability to independently predict how much you'll enjoy something. We are, scientifically speaking, very bad at this. But if something reliably makes others happy, it's likely to make you happy, too. Weight other people's opinions and user reviews heavily in your purchasing decisions.

Happiness is a lot harder to come by than money. So when you *do* spend money, keep these eight lessons in mind to maximize whatever happiness it can buy for you. And remember: **it's science!**

Lived Fast, Died Young, Left a Tired Corpse

It's easy to forget just **how crazy things got during the Web 1.0 bubble in 2000**. That was over ten years ago. For context, [Mark Zuckerberg](#) was all of *sixteen* when the original web bubble popped.



There are two films which captured the hyperbole and excess of the original dot com bubble especially well.



The first is the documentary [Startup.com](#). It's about the prototypical web 1.0 company: one predicated on an idea that made absolutely no sense, which proceeded to flame out in a spectacular and all too typical way for the era. This one just happened to occur on

digital film. The govworks.com website described in the documentary, the one that burned through \$60 million in 18 months, is now one of those ubiquitous domain squatter pages. A sign of the times, perhaps.

The second film was one I had always wanted to see, but wasn't able to until a few days ago: [Code Rush](#). For a very long time, Code Rush was almost impossible to find, but [the activism of Andy Baio](#) nudged the director to make the film available under Creative Commons. You can now watch it online — and you absolutely should.

Remember when people *charged money for a web browser*? That was Netscape.

Code Rush is a PBS documentary recorded at Netscape from 1998-1999, focusing on the [open sourcing of the Netscape code](#). As the documentary makes painfully clear, this wasn't an act of strategy so much as an act of desperation. That's what happens when the company behind the world's most ubiquitous operating system decides a web browser should be a standard part of the operating system.

Everyone in the documentary knows they're doomed; in fact, the phrase "we're doomed" is a common refrain throughout the film. But despite the gallows humor and the dark tone, parts of it are oddly inspiring. These are engineers who are working heroic, impossible schedules for a goal they're not sure they can achieve — or that they'll even survive as an organization long enough to even finish.

The most vivid difference between Startup.com and Code Rush is that Netscape, despite all their other mistakes and missteps, didn't just burn through millions of dollars for no discernable reason. They produced a **meaningful legacy**:

- Through Netscape Navigator, the original popularization of HTML and the internet itself.
- With the release of the Netscape source code on March 31st, 1998, the unlikely birth of the commercial open source movement.
- Eventually producing the first credible threat to Internet Explorer in the form of Mozilla Firefox 1.0 in 2004.

Do you want money? Fame? Job security? Or do you want to change the world ... eventually? Consider how many legendary hackers went on to brilliant careers from Netscape: Jamie Zawinski, Brendan Eich, Stuart Parmenter, Marc Andreessen. The lessons of Netscape live on, even though the company doesn't. Code Rush is ultimately a **meditation on the meaning of work as a programmer**.

As Startup.com and Code Rush illustrate, the hard part is figuring out *why* you are working all those long hours. Consider carefully, lest the arc of your career mirror that of so many failed tech bubble companies: **lived fast, died young, left a tired corpse.**

 **Tom Foremski@tomforemski**

“On his deathbed, did Steve Jobs regret all the time he spent at the office?”

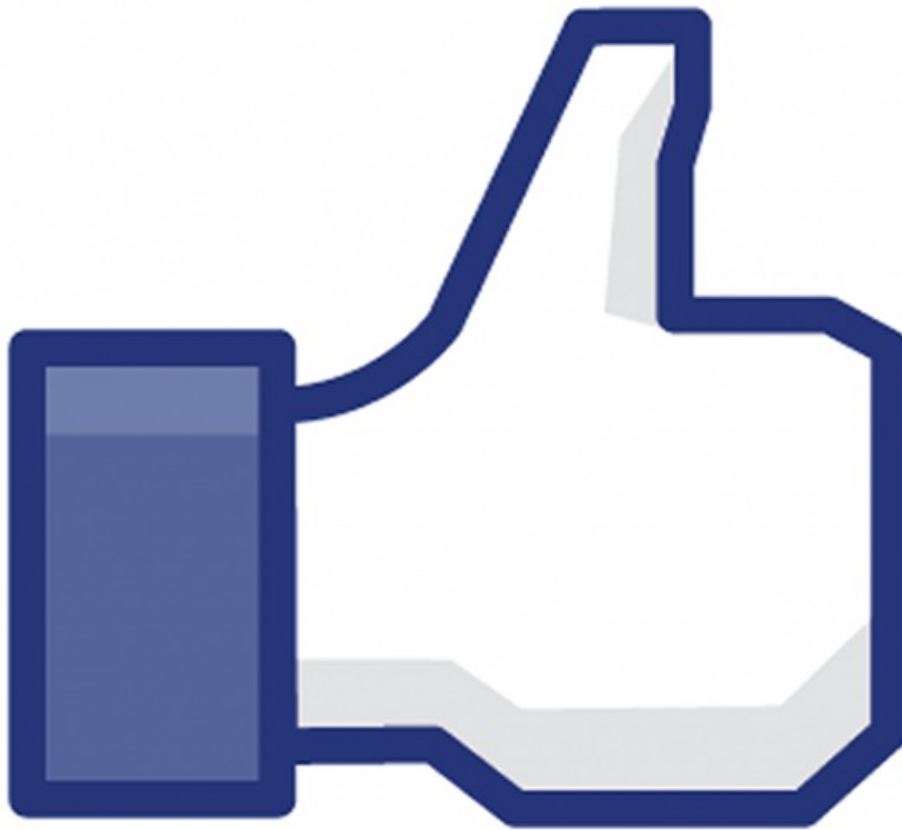
8:45 PM - 5 Oct 11

Gift this book to your friends...for [free](#).

Become a Hyperink reader. Get a [special surprise](#).

Like the book? Support our author and leave a [comment](#)!

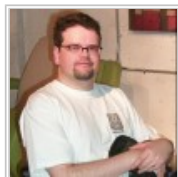
Like this book?



[Click to share a free copy with your Facebook friends!](#)

(Don't worry, it won't auto share. And if you're reading on Kindle or Nook, this is an easy way to get a free PDF copy for yourself!)

About *The* Author



Jeff Atwood

I'm Jeff Atwood. I live in Berkeley, CA with my wife, two cats, one three children, and a whole lot of computers. I was weaned as a software developer on various implementations of Microsoft BASIC in the 80's, starting with my first microcomputer, the Texas Instruments TI-99/4a. I continued on the PC with Visual Basic 3.0 and Windows 3.1 in the early 90's, although I also spent significant time writing Pascal code in the first versions of Delphi. I am now quite comfortable in VB.NET or C#, despite the evils of case sensitivity. I'm currently learning Ruby.

I consider myself a reasonably experienced Windowsweb software developer with a particular interest in the human side of software development, as represented in my recommended developer reading list. Computers are fascinating machines, but they're mostly a reflection of the people using them. In the art of software development, studying code isn't enough; you have to study the people behind the software, too.

In 2004 I began Coding Horror. I don't mean to be overly dramatic, but it changed my life. Everything that comes after was made possible by this blog.

In 2005, I found my dream job at Vertigo Software and moved to California. You can take a virtual tour of my old office if you'd like.

In 2008 I decided to choose my own adventure. I founded and built stackoverflow.com, and what would ultimately become the Stack Exchange network of Q&A sites, in a joint venture with Joel Spolsky. The Stack Exchange network is now one of the top 150 largest sites on the Internet.

In early 2012 I decided to leave Stack Exchange and spend time with my growing family while I think about what the next thing could be.

Content © 2012 Jeff Atwood. Logo image used with permission of the author. © 1993 Steven C. McConnell. All Rights Reserved.

About the Publisher

Hyperink is the easiest way for anyone to publish a beautiful, high-quality book.

We work closely with subject matter experts to create each book. We cover topics ranging from higher education to job recruiting, from Android apps marketing to barefoot running.

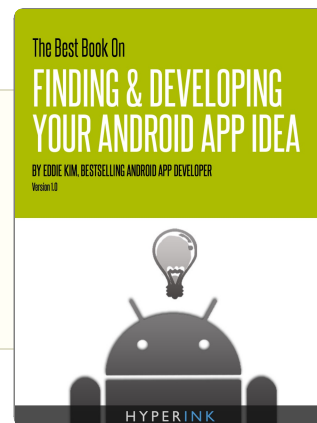
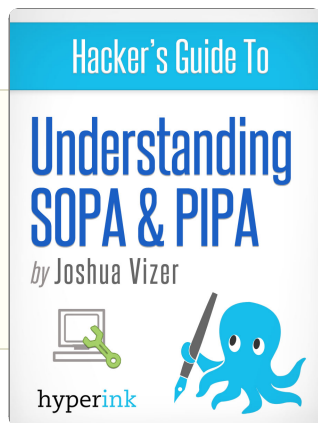
If you have interesting knowledge that people are willing to pay for, especially if you've already produced content on the topic, please [reach out](#) to us! There's no writing required and it's a unique opportunity to build your own brand and earn royalties.

Hyperink is based in SF and actively hiring people who want to shape publishing's future. [Email us](#) if you'd like to meet our team!

Note: If you're reading this book in print or on a device that's not web-enabled, **please email** books@hyperinkpress.com with the title of this book in the subject line. We'll send you a PDF copy, so you can access all of the great content we've included as clickable links.

Get in touch:   

Other **A**wesome Books



Hyperink Benefits

★ Interesting Insights ★ The Best Commentary ★ Shocking Trivia

- Hacker's Guide To Understanding SOPA and PIPA
- CityVille: Pro Gaming Tips
- The Space Elevator Project
- Pro Gaming Tips: Farmville
- The Best Book on Finding & Developing Your Android App Idea
- Lulzsec
- How To Get The Most From Your Kindle Device
- How to Build a Huge Following on Pinterest
- Pro Gaming Tips: Halo Reach
- The Best Book on Designing iPhone and iPad Apps

Use code **INKYBUCKS1** to save 25% off your next purchase. [Click here!](#) →

Copyright © 2012-Present. Hyperink Inc.

The standard legal stuff:

All rights reserved. No part of this book may be reproduced in any form or by any electronic or mechanical means, including information storage and retrieval systems, without permission in writing from Hyperink Inc., except for brief excerpts in reviews or analysis.

Our note:

Please don't make copies of this book. We work hard to provide the highest quality content possible - and we share a lot of it for free on our sites - but these books are how we support our authors and the whole enterprise. You're welcome to borrow (reasonable) pieces of it as needed, as long as you give us credit.

Thanks!

The Hyperink Team

Disclaimer

This ebook provides information that you read and use at your own risk. This book is not affiliated with or sponsored by any other works, authors, or publishers mentioned in the content.

Thanks for understanding. Good luck!