

BLOG

TO BOOK



JEFF ATWOOD

How to Stop Sucking
and Be Awesome Instead



codinghorror.com

Table of Contents

How to Suck Less

Today's Not

Today Is Goof Off at Work Day

But You Did Not Persuade Me

The Only Truly Failed Project

Who Needs Talent When You Have Intensity?

Are You An Expert?

On Our Project, We're Always 90 Percent Done

Managing with Trust

Boyd's Law of Iteration

Overnight Success: It Takes Years

Programming

How to Become a Better Programmer by Not Programming

The Broken Window Theory

Programming: Love It or Leave It

Some Lessons from Forth

The Joy of Deletion

Separating Programming Sheep from Non-Programming Goats

Are You Following the Instructions on the Paint Can?

Curly's Law: Do One Thing

The Ultimate Code Kata

In Programming, One Is the Loneliest Number

Who's Your Coding Buddy?

Software Apprenticeship

Web Design Principles

Judging Websites

In Pursuit of Simplicity

Will Apps Kill Websites?

Doing It Like Everybody Else

The One-Button Mystique

Usability on the Cheap and Easy

The Opposite of Fitts' Law

Usability vs. Learnability

[Google's Number One UI Mistake](#)

[But It's Just One More](#)

[Just Say No](#)

[UI Is Hard](#)

[Testing](#)

[Good Test/Bad Test](#)

[Unit Testing vs. Beta Testing](#)

[Sometimes It's a Hardware Problem](#)

[Exception-Driven Development](#)

[Know Your User](#)

[The Rise and Fall of Homo Logicus](#)

[Ivory Tower Development](#)

[This Is What Happens When You Let Developers Create UI](#)

[Defending Perpetual Intermediacy.](#)

[Every User Lies](#)

[Shipping Isn't Enough](#)

[Don't Ask—Observe](#)

[Are Features the Enemy?](#)

[The Organism Will Do Whatever It Damn Well Pleases](#)

[For a Bit of Colored Ribbon](#)

[Building Social Software for the Anti-Social](#)

[Causes We Should Care About](#)

[Preserving the Internet... And Everything Else](#)

[The Importance of Net Neutrality.](#)

[Youtube vs. Fair Use](#)

[Gaming](#)

[Everything I Needed to Know About Programming I Learned from BASIC](#)

[Programming Games, Analyzing Games](#)

[Game Player, Game Programmer](#)

[Things to Read](#)

[Programmers Don't Read Books, But You Should](#)

[Nobody's Going to Help You, and That's Awesome](#)

[Computer Crime, Then and Now](#)

[How to Talk to Human Beings](#)

[Practicing the Fundamentals: The New Turing Omnibus](#)

OceanofPDF.com

I.

How to Suck Less

OceanofPDF.com

Todon't

What do you need to do today? Other than read this blog entry, I mean.

Have you ever noticed that a huge percentage of [Lifehacker](#)-like productivity porn site content is a breathless description of the details of **Yet Another To-Do Application**? There are dozens upon dozens of the things to choose from, on any platform you can name. At this point it's getting a little ridiculous; per [Lifehacker's Law](#), you'd need a to-do app just to keep track of all the freaking to-do apps.



I've tried to maintain to-do lists at various points in my life. And I've always failed. Utterly and completely. Even turning it into a game, like the cleverly constructed [Epic Win app](#), didn't work for me.



[Watch: EpicWin: Pre-Release Trailer](#)

Eventually I realized that the problem wasn't me. All my to-do lists started out as innocuous tools to assist me in my life, but slowly transformed, each and every time, into thankless, soul-draining exercises in reductionism. [My to-do list was killing me](#). Adam Wozniak nails it:

1. Lists give the illusion of progress.
2. Lists give the illusion of accomplishment.
3. Lists make you feel guilty for not achieving these things.
4. Lists make you feel guilty for continually delaying certain items.
5. Lists make you feel guilty for not doing things you don't want to be doing anyway.
6. Lists make you prioritize the wrong things.
7. Lists are inefficient. (Think of what you could be doing with all the time you spend maintaining your lists!)

8. Lists suck the enjoyment out of activities, making most things feel like an obligation.
9. Lists don't actually make you more organized long term.
10. Lists can close you off to spontaneity and exploration of things you didn't plan for. (Let's face it, it's impossible to really plan some things in life.)

For the things in my life that actually mattered, I've never needed any to-do list to tell me to do them. If I did, then that'd be awfully strong evidence that I have some serious life problems to face before considering the rather trivial matter of which to-do lifehack fits my personality best. As for the things that didn't matter in my life, well, those just tended to pile up endlessly in the old to-do list. And **the collective psychic weight of all these minor undone tasks** were caught up in my ever-growing to-do katamari ball, where they continually weighed on me, day after day.

Yes, there's that ever-present giant to-do list, hanging right there over your head like a guillotine, growing [sharper and heavier every day](#).

“Like a crazy hoarder I mistake the root cause of my growing mountain of incomplete work. The hoarder thinks he has a storage problem when he really has a 'throwing things away problem'. I say I am 'time poor' as if the problem is that poor me is given only 24 hours in a day. It's more accurate to say... what exactly? It seems crazy for a crazy person to use his own crazy reasoning to diagnose his own crazy condition. Maybe I too easily add new projects to my list, or I am too reluctant to exit from unsuccessful projects. Perhaps I am too reluctant to let a task go, to ship what I've done. They're never perfect, never good enough.

“And I know I'm not alone in making the easy claim that I am 'time poor'. So many people claim to be time poor, when really we are poor at prioritizing, or poor at decisiveness, or don't know how to say 'no' (...to other people, to our own ideas).

“If only I had a hidden store of time, or if only I had magical organisation tools, or if only I could improve my productive throughput, then, only then would I be able to get things done, to consolidate the growing backlogs and

todo lists into one clear line of work, and plough through it like an arctic ice breaker carving its way through a sheet of ice.”

But are you using the right guillotine? Maybe it'd work better if you tried this newer, shinier guillotine? I'd like to offer you some advice:

1. There's only one, and exactly one, item anyone should ever need on their to-do list. Everything else is superfluous.
2. You shouldn't have a to-do list in the first place.
3. Declare to-do bankruptcy right now. Throw out your to-do list. It's hurting you.
4. Yes, seriously.
5. Maybe it is a little scary, but the right choices are always a little scary, so do it anyway.
6. No, I wasn't kidding.
7. Isn't Hall and Oates awesome? I know, rhetorical question. But still.
8. Look, this is becoming counterproductive.
9. Wait a second, did I just make a list?

Here's my challenge. **If you can't wake up every day and, using your 100 percent original equipment, God-given organic brain, come up with the three most important things you need to do that day**—then you should seriously work on fixing that. I don't mean install another app, or read more productivity blogs and books. You have to figure out what's important to you and what motivates you; ask yourself why that stuff isn't gnawing at you enough to make you get it done. Fix that.

Tools will come and go, but **your brain and your gut will be here with you for the rest of your life**. Learn to trust them. And if you can't, do whatever it takes to train them until you can trust them. If it matters, if it really matters, you'll remember to do it. And if you don't, well, maybe you'll get to it one of these days. Or not. And that's cool too.

Today Is Goof Off at Work Day

When you're hired at Google, you only have to do the job you were hired for 80 percent of the time. The other 20 percent of the time, you can work on whatever you like—provided it advances Google in some way. At least, that's the theory.

Google's 20 percent time policy is well known in software engineering circles by now. What's not as well known is that this concept [dates all the way back to 1948 at 3M](#).

“In 1974, 3M scientist Art Fry came up with a clever invention. He thought if he could apply an adhesive (dreamed up by colleague Spencer Silver several years earlier) to the back of a piece of paper, he could create the perfect bookmark, one that kept place in his church hymnal. He called it the Post-It Note. Fry came up with the now iconic product (he talks to the Smithsonian about it [here](#)) during his “15 percent time,” a program at 3M that allows employees to use a portion of their paid time to chase rainbows and hatch their own ideas. It might seem like a squishy employee benefit. But the time has actually produced many of the company's best-selling products and has set a precedent for some of the top technology companies of the day, like Google and Hewlett-Packard.”

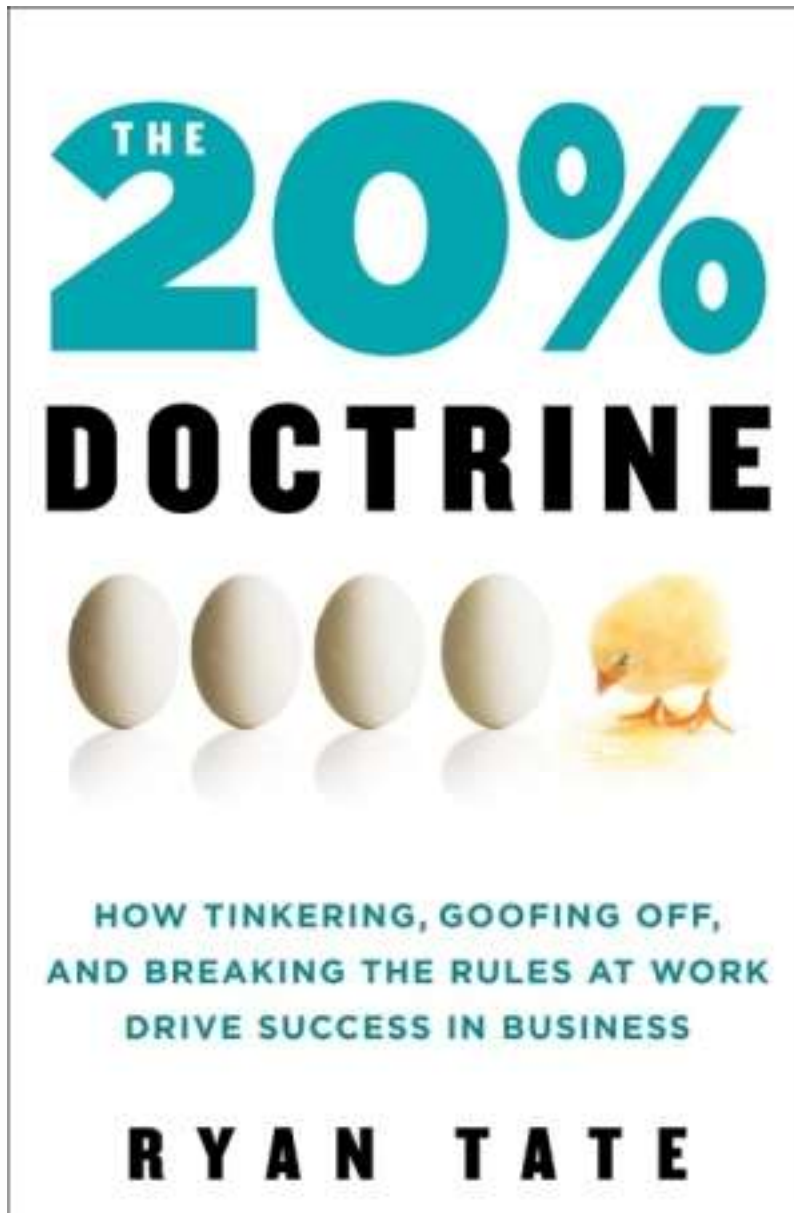
There's not much documentation on HP's version of this; when I do find mentions of it, it's always referred to as a “convention,” not an explicit policy. Robert X. Cringely [provides more detail](#):

“Google didn’t invent that: HP did. And the way the process was instituted at HP was quite formal in that the 10 percent time was after lunch on Fridays. Imagine what it must have been like on Friday afternoons in Palo Alto with every engineer working on some wild-ass idea. And the other part of the system was that those engineers had access to what they called ‘lab stores’—anything needed to do the job, whether it was a microscope or a magnetron or a barrel of acetone could be taken without question on Friday afternoons from the HP warehouses. This enabled a flurry of innovation that produced some of HP’s greatest products including those printers.”

Maybe HP did invent this, since they've been around since 1939. Dave Raggett, for example, apparently [played a major role in inventing HTML](#) on his 10 percent time at HP.

Although the concept predates Google, they've done more to validate it as an actual strategy and popularize it in tech circles than anyone else. Oddly enough, I can't [find](#) any mention of the 20 percent time benefit listed on the [current Google jobs page](#), but it's an integral part of Google's culture. And for good reason: notable 20 percent projects include [GMail](#), Google News, Google Talk, and AdSense. According to ex-employee Marissa Meyer, [as many as half](#) of Google's products originated from that 20 percent time.

At Hewlett-Packard, 3M, and Google, "many" of their best and most popular products come from **the thin sliver of time they granted employees to work on whatever they wanted to**. What does this mean? Should we all be goofing off more at work and experimenting with our own ideas? That's what the book [The 20 percent Doctrine](#) explores.



Closely related to 20 percent time is the **Hack Day**. Hack Days carve out a specific 24 hour timeframe from the schedule, encouraging large groups to come together to work collaboratively (or in friendly competition) during that period. Chad Dickerson [instituted one of the first](#) at Yahoo in 2005.

“The Friday before, I had organized the first internal Hack Day at Yahoo! with the help of a loosely-organized band of people around the company. The ‘hack’ designation for the day was a tip of the hat to hacker culture, but also a nod to the fact that we were trying to fix a system that didn’t work particularly well. The idea was really simple: all the engineers in our

division were given the day off to build anything they wanted to build. The only rules were to build something in 24 hours and then show it at the end of the period. The basic structure of the event itself was inspired by what we had seen at small startups, but no one had attempted such an event at a large scale at an established company.”

“The first Yahoo! Hack Day was clearly a success. In a company that was struggling to innovate, about seventy prototypes appeared out of nowhere in a single 24-hour period and they were presented in a joyfully enthusiastic environment where people whooped and yelled and cheered. Sleep-deprived, t-shirt-clad developers stayed late at work on a Friday night to show prototypes they had built for no other reason than they wanted to build something. In his seminal book about open source software, *The Cathedral and the Bazaar*, Eric Raymond wrote: ‘Every good work of software starts by scratching a developer’s personal itch.’ There clearly had been a lot of developer itching around Yahoo! but it took Hack Day to let them issue a collective cathartic scratch.”

Atlassian's version, a [quarterly ShipIt Day](#), also dates back to 2005. Interestingly, they also attempted to emulate Google's 20 percent time policy with [mixed results](#).

“Far and away, the biggest problem was scheduling time for 20 percent work. As one person put it, ‘Getting 20 percent time is incredibly difficult amongst all the pressure to deliver new features and bug fixes.’ Atlassian has frequent product releases, so it is very hard for teams to schedule ‘down time’. Small teams in particular found it hard to afford time away from core product development. This wasn’t due to Team Leaders being harsh. It was often due to developers not wanting to increase the workload on their peers while they did 20 percent work. They like the products they are developing and are proud of their efforts. However, they don’t want to be seen as enjoying a privilege while others carry the workload.”

I think there's enough of a track record of documented success that **it's worth lobbying for something like Hack Days or 20 percent time wherever you work**. But before you do, consider if you and your company are ready:

1. Is there adequate slack in the schedule?

You can't realistically achieve 20 percent time, or even a single measly hack day, if there's absolutely zero slack in the schedule. If everyone around you is working full-tilt boogie as hard as they can, all the time, that's... probably not healthy. Sure, everyone has crunch times now and then, but if your work environment feels like constant crunch time, you'll need to deal with that first. For ammunition, try [Tom Demarco's book, "Slack."](#)

2. Does daydreaming time matter?

If anyone gets flak for not "looking busy," your company's work culture may not be able to support an initiative like this. There has to be buy-in at the pointy-haired-boss level that time spent thinking and daydreaming is a valid part of work. Daydreaming is not the antithesis of work; on the contrary, creative problem solving [requires it](#).

3. Is failure accepted?

When given the freedom to "work on whatever you want," the powers that be have to really mean it for the work to matter. Mostly that means providing employees the unfettered freedom to fail miserably at their skunkworks projects, sans repercussion or judgment. Without failure, and lots of the stuff, there can be no innovation, or true experimentation. The value of (quickly!) learning from failures and moving on is enormous.

4. Is individual experimentation respected?

If there isn't a healthy respect for individual experimentation versus the neverending pursuit of the Next Thing on the collective project task list, these initiatives are destined to fail. You have to truly believe, as a company, and as peers, that crucial innovations and improvements can come from everyone at the company at any time, in bottom-up fashion—they aren't delivered from on high at scheduled release intervals in the almighty Master Plan.

Having some official acknowledgement that time spent working on whatever you think will make things better around these here parts is not just tolerated—but encouraged—might go a long way towards **making work feel a lot less like work**.

But You Did Not Persuade Me

One of my favorite movie scenes in recent memory is from [The Last King of Scotland](#), a dramatized "biography" of the megalomaniac dictator [Idi Amin](#), as seen through the eyes of a fictional Scottish personal physician.



Idi Amin: "I Want you to tell me what to do!"

Garrigan: "You want ME to tell YOU what to do?"

Amin: "Yes, you are my advisor. You are the only one I can trust in here. You should have told me not to throw the Asians out in the first place!"

Garrigan: "I did!"

Amin: "But you did not persuade me, Nicholas! You did not persuade me!"

If you haven't watched this movie yet, you should. It is amazing.

What I love about this tour de force of a scene—beyond the incredible acting—is that it illustrates just how powerful of a force persuasion really is. In the

hands of a madman or demagogue, dangerously powerful. Hopefully you don't deal with too many insane dictators on a daily basis, but the reason this scene works so well is the unavoidable truth it exposes: **to have any hope of influencing others, you must be able to persuade them.**

Steve Yegge is as accomplished a software engineer as I can think of. I was amazed to hear him tell us repeatedly and at length on a podcast that [the one thing every software engineer should know](#) is not how to write amazing code, but how to market themselves and their project. What is marketing except persuasion?

Marc Hedlund, who founded Wesabe and is now the VP of Engineering at Etsy, thinks of himself not as a CEO or boss, but as the [Lobbyist-in-Chief](#). I believe that could be rewritten as Persuader-in-Chief with no loss of meaning or nuance.

“I was recently asked how I run our development team. I said, ‘Well, basically I blog about something I think we should do, and if the blog post convinces the developers, they do it. If not, I lobby for it, and if that fails too, the idea falls on the floor. They need my approval to launch something, but that’s it. That’s as much ‘running things’ as I do, and most of the ideas come from other people at this point, not from me and my blog posts. I’ve argued against some of our most successful ideas, so it’s a good thing I don’t try to exert more control.’

“I’m exaggerating somewhat; of course I haven’t blogged about all of our ideas yet. But I do think of myself as Lobbyist-in-Chief, and I have lots of good examples of cases where I failed to talk people into an idea and it didn’t happen as a result. One person I said this to asked, ‘So who holds the product vision, then?’ and I replied, ‘Well, I guess I do,’ but really that’s not right. We all do. The product is the result of the ideas that together we’ve agreed to pursue. I recruit people based on their interest in and enthusiasm about the ideas behind Wesabe, and then set them loose, and we all talk and listen constantly. That’s how it works—and believe it or not, it does work.”

So how do we persuade? Primarily, I think, when we [lead by example](#). Even if that means getting down on your knees and [cleaning a toilet](#) to show someone else how it's done. But maybe you're not a leader. Maybe you're just a lowly peon. Even as a peon, it's still possible to [persuade your team](#)

[and those around you](#). A commenter summarized this grassroots method of persuasion nicely:

- His ideas were, on the whole, pretty good.
- He worked mostly bottom-up rather than top-down.
- He worked to gain the trust of others first by dogfooding his own recommendations before pushing them on others.
- He was patient and waited for the wheels to turn.

Science and data are among the best ways to be objectively persuasive, but remember that data alone isn't the reductionist end of every single topic. Beware the [41 shades of blue](#) pitfall.

“Yes, it’s true that a team at Google couldn’t decide between two blues, so they’re testing 41 shades between each blue to see which one performs better. I had a recent debate over whether a border should be 3, 4 or 5 pixels wide, and was asked to prove my case. I can’t operate in an environment like that. I’ve grown tired of debating such minuscule design decisions. There are more exciting design problems in this world to tackle.”

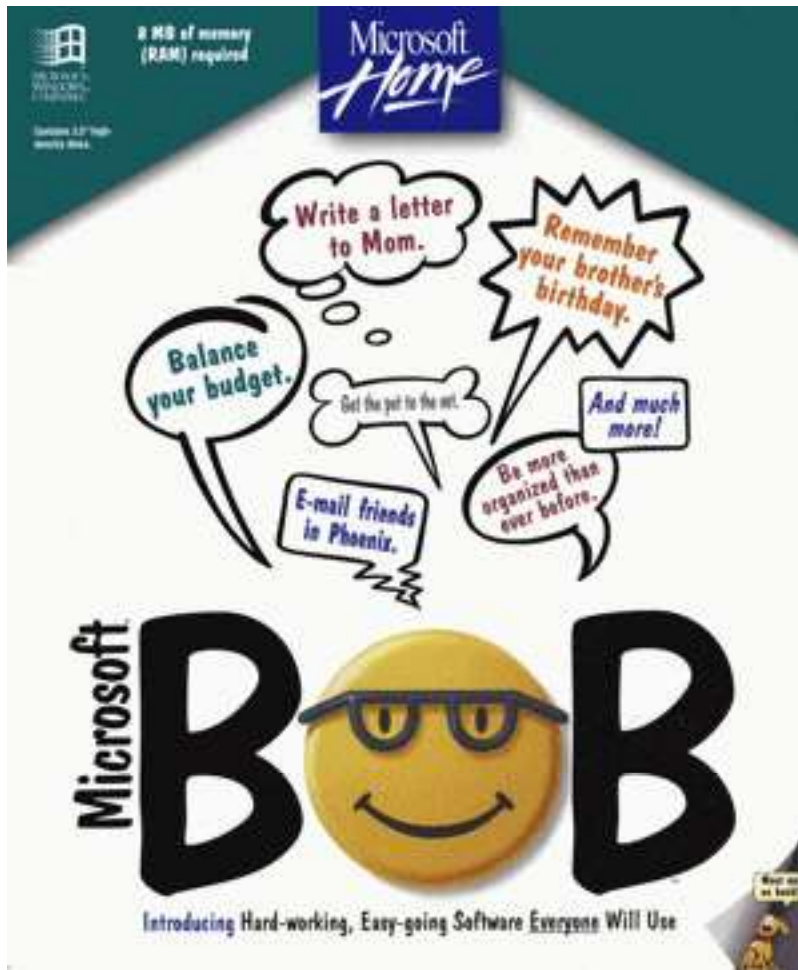
If I measure by click data alone, [all Internet advertising should have breasts in it](#). Incorporate data, by all means. But you need to tell a bigger, grander, more inspiring story than that to be truly persuasive.

I [re-read “Letter from a Birmingham Jail”](#) every year because I believe it is the single best persuasive essay I've ever read. It is remarkably persuasive without ever resorting to anger, incivility, or invective. [Read it now](#). But do more than just read; study it. How does it work? Why does it work? Does it cite any data? What techniques make this essay so incredibly compelling?

Nobody ever changed anything by remaining quiet, idly standing by, or blending into the faceless, voiceless masses. If you ever want to effect change, in your work, in your life, **you must learn to persuade others**.

The Only Truly Failed Project

Do you remember [Microsoft Bob](#)? If you do, you probably remember it as an intensely marketed but laughable failure—what some call [the "number one flop" at Microsoft](#).




With personalized Bob, you can have the convenience of desktop software without the hassle of installing it. 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!

You don't need a manual. Your personal guide is always around to help you if you need it!



- 1. Calendar**
Help you stay on top of the day, so you can make sure you don't miss any important dates. And you can even set reminders.
- 2. Letter Writer**
Write a letter to a friend, family member, or even a business. It's so easy, you can be writing one in less than 10 minutes.
- 3. Checklist**
Remember to do your laundry, mow the lawn, and take the dog to the vet. The Bob Checklist helps you stay on top of everything.
- 4. Address Book**
Keep a list of all the people you know. You can even add phone numbers, email addresses, and more. It's so easy to use, you can be adding names in less than 10 minutes.
- 5. E-Mail**
Send an email to your friends, family, or even a business. It's so easy, you can be sending one in less than 10 minutes.
- 6. Household Manager**
Remember to do your laundry, mow the lawn, and take the dog to the vet. The Bob Checklist helps you stay on top of everything.
- 7. Internet**
Explore a world of information. It's so easy, you can be exploring in less than 10 minutes.
- 8. Household Manager**
Remember to do your laundry, mow the lawn, and take the dog to the vet. The Bob Checklist helps you stay on top of everything.

Make yourself at home. Customize your "Home" with address books, notes, and fun objects. Keep your files private—or share information with others. Even more programs for MSN 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're even here to help you like to work!

These eight easy-going, hard-working programs are instant. In just one click, 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, due dates from your Checklist, or birthdays from your Address Book. Bob makes hard-to-keep things simple for you!

© Microsoft Corporation
0295 Part No. 64035

Microsoft

There's no question that Microsoft Bob was nothing short of an unmitigated disaster. But that's the funny thing about failures—they often lead to later successes. Take it from someone who [lived and breathed the Bob project](#):

“I was the one who sent Bill Gates email at the height of the positive Bob-mania that said we were likely to face a horrible backlash. Tech influentials had started telling me that they were going to bury Bob. They not only didn't like it, they were somehow angry that it had even been developed. It was personal.

“And that's exactly what happened. Bob got killed. But first, it was ridiculed and stomped.

“For Microsoft, it was a costly mistake. For the people who worked on it, Bob taught many lessons. Lessons that came into play for subsequent products that made a big impact, both at Microsoft and beyond.

“How many people know that the lead developer for Bob 2.0 was also the [co-founder of Valve](#) and the development lead for Half-Life, which became an industry phenomenon, winning more than 50 Game of the Year awards and selling more than 10 million copies?

“Or that Darrin Massena—development lead for Bob 1.0, most recently named Technical Innovator of the Year here in Washington State—and Valve co-founder [Mike Harrington](#) are the co-founders and partners behind [Picnik](#)—which is now the world's leading online photo editor, attracting almost 40 million visits a month and a million unique users a day.”

And then, of course, I'd be remiss if I didn't mention that Melinda French—Bill Gates' [future wife](#)—managed the Microsoft Bob project at one point. Bob was the first Microsoft consumer project that [Bill Gates personally had a hand in launching](#). Well, at least he got a wife out of it.

Yes, Bob was an obvious, undisputed, and epic failure. We can point and laugh at Bob. But to me, **Bob is less of a comic figure than a tragic one.**

Unless you're an exceptionally lucky software developer, you've probably worked on more projects that failed than projects that succeeded. Failure is [de rigueur in our industry](#). Odds are, you're working on a project that will fail right now. Oh sure, it may not seem like a failure yet. Maybe it'll fail in some completely unanticipated way. Heck, maybe your project will buck the odds and even succeed.

But I doubt it.

I [own a boxed copy of Microsoft Bob](#). I keep it on my shelf to remind me that these kinds of relentless, inevitable failures aren't the crushing setbacks they often appear from the outside. On the contrary; I believe it's [impossible to succeed without failing](#).

“Charles Bosk, a sociologist at the University of Pennsylvania, once conducted a set of interviews with young doctors who had either resigned or been fired from neurosurgery-training programs, in an effort to figure out what separated the unsuccessful surgeons from their successful counterparts.

“He concluded that, far more than technical skills or intelligence, what was necessary for success was the sort of attitude that Quest has—a practical-minded obsession with the possibility and the consequences of failure. ‘When I interviewed the surgeons who were fired, I used to leave the interview shaking,’ Bosk said. ‘I would hear these horrible stories about what they did wrong, but the thing was that they didn't know that what they did was wrong. In my interviewing, I began to develop what I thought was an indicator of whether someone was going to be a good surgeon or not. It was a couple of simple questions: Have you ever made a mistake? And, if so, what was your worst mistake? The people who said, ‘Gee, I haven't really had one,’ or, ‘I've had a couple of bad outcomes but they were due to things outside my control’—invariably those were the worst candidates. And the residents who said, ‘I make mistakes all the time. There was this horrible thing that happened just yesterday and here's what it was.’ They were the best. They had the ability to rethink everything that they'd done and imagine how they might have done it differently.’”

I recently watched the documentary “[Tilt: The Battle to Save Pinball.](#)”



[Watch: Trailer for “TILT: The Battle To Save Pinball”](#)

It's a gripping story of a pinball industry in crisis. In order to save it, the engineers at Williams—the only remaining manufacturer of pinball machines in the United States—were given a herculean task: invent a new

form of pinball so compelling that it makes all previous pinball machines seem obsolete. I don't want to spoil the whole documentary, so I'll gloss over exactly how that happened, but astoundingly enough—they succeeded.

And then were promptly laid off en masse, as Williams shut down its pinball operations.

Unlike Microsoft Bob, the Williams engineers built an almost revolutionary product that was both critically acclaimed and sold well—but **none of that mattered**. It's sobering to watch the end reel of “Tilt,” as the engineers involved mournfully discuss the termination of their bold and seemingly successful project.

“Everyone was in awe. They couldn't understand why it happened. Here we'd just done this thing that from all we could tell was a total success. Why would they do that?”

“We succeeded. Management gave us an impossible goal, and we sat there and we actually did what they thought we couldn't do.

“You know, we didn't really win... we lost. I gave it everything I had. I think that those fifty guys that worked on it, they also passionately did everything that they could.”

Sometimes, **even when your project succeeds, you've failed**. Due to forces entirely beyond your control. It's depressing, but it's reality.

The trail out isn't all doom and gloom. It also documents the ways in which these talented pinball engineers went on to practice their craft after being laid off. Most of them still work in the video game or pinball industry. Some freelance. Others formed their own companies. A few went on to work at Stern Pinball, which figured out how to make a small number of pinball machines and still turn a profit.

These two stories, these two projects—the abject failure of Microsoft Bob, and the aborted success of Pinball 2000—have something in common beyond mere failure. All the engineers involved **not only survived these failures, but often went on to greater success afterwards**. Possibly as a direct result of their work on these “failures.”

Failure is a wonderful teacher. But there's no need to seek out failure. It will find you. Whatever project you're working on, consider it an opportunity to learn and practice your craft. [It's worth doing because, well, it's worth doing](#). The journey of the project should be its own reward, regardless of whatever happens to lie at the end of that journey.

The only truly failed project is **the one where you didn't learn anything along the way.**

OceanofPDF.com

Who Needs Talent When You Have Intensity?

Jack Black, in the DVD extras for “[School of Rock](#),” had this to say in an interview:

“I had to learn how to play electric guitar a little bit because all I play is acoustic guitar. And I'm still not very good at electric guitar. And the truth is, I'm not very good at acoustic guitar, but I make up for it with intensity.”

It's hard to appreciate how true this is until you've heard (or better yet, seen) Jack Black's band [Tenacious D](#) perform. Musically, they're terrible. But they still manage to be thoroughly entertaining and [often hilarious](#).



I was reminded of this Jack Black quote while reading ["It's not about you"](#) in the excellent Creating Passionate Users blog:

“The I-don't-matter-so-don't-introduce-myself plan was just the beginning of the ‘it's not about YOU’ experiment. I would conduct the rest of the five day course with all of my energy devoted to making THEM smarter, rather than trying to make sure they knew how smart I was. (A clever and necessary strategy on my part, since I'm not all that smart.)

“The year-long experiment was a success, and I won a nice bonus from Sun for being one of only four instructors in north America to get the highest possible customer evaluations. But what was remarkable about this is that this happened in spite of my not being a particularly good instructor or Java guru. I proved that a very average instructor could get exceptional results by putting the focus ENTIRELY on the students. I paid no attention to whether they thought I knew my stuff.

“And when I say that I was average, that's really a stretch. I have almost no presentation skills. When I first started at Sun I thought I was going to be fired because I refused to ever use the overhead slides and just relied on the whiteboard (where I drew largely unrecognizable objects and unreadable code). But... I say average when you evaluate me against a metric of traditional stand-up instructor presentation skills. Which I believe are largely bullshit anyway. Assuming you meet some very minimal threshold for teaching, all that matters is that you help the students become smarter. You help them learn... by doing whatever it takes. And that usually has nothing to do with what comes out of your mouth, and has everything to do with what happens between their ears. You, as the instructor, have to design and enable situations that cause things to happen. Exercises, labs, debates, discussions, heavy interaction. In other words, things that THEY do, not things that YOU do (except that you create the scenarios).”

These inspiring results echo my feelings about what it takes to be a "good" programmer. Don't be cowed by the existence of thousands of developers far more talented than you are. **Who needs talent when you have intensity?**

Are You An Expert?

I think I have a problem with authority. Starting with [my own](#).

“It troubles me greatly to hear that people see me as an expert or an authority, and not a fellow amateur.

“If I've learned anything in my career, it is that approaching software development as an expert, as someone who has already discovered everything there is to know about a given topic, is the one surest way to fail.

“Experts are, if anything, more suspect than the amateurs, because they're less honest. You *should* question everything I write here, in the same way you question everything you've ever read online—or anywhere else for that matter. Your own research and data should trump any claims you read from anyone, no matter how much of an authority or expert you, I, Google, or the general community at large may believe them to be.”

Have you ever worked with software developers who [thought of themselves as experts](#), with almost universally painful results? I certainly have. You might say I've developed **an anti-expert bias**. Apparently, so has Wikipedia; a section titled [warnings to expert editors](#) explains:

1. Experts can identify themselves on their user page and list whatever credentials and experience they wish to publicly divulge. It is difficult to maintain a claim of expertise while being anonymous. In practice, there is no advantage (and considerable disadvantage) in divulging one's expertise in this way.
2. Experts do not have any other privileges in resolving edit conflicts in their favor: in a content dispute between a (supposed) expert and a non-expert, it is not permissible for the expert to "pull rank" and declare victory. In short, "Because I say so" is never an acceptable justification for a claim in Wikipedia, regardless of expertise. Likewise, expert contributions are not protected from subsequent revisions from non-experts, nor is there any mechanism to do so. Ideally, if not always in practice, it is the quality of the edits that counts.

3. There is a **strong undercurrent of anti-expert bias in Wikipedia**. Thus, if you become recognized as an expert you will be held to higher standards of conduct than non-experts.

Let's stop for a moment to savor the paradox of a free and open encyclopedia written by people who [view the contributions of experts with healthy skepticism](#). How could that possibly work?

I'd argue that's the only way it could work—when all contributions are viewed critically, regardless of source. This is a radical inversion of power. But a radical inversion of power is exactly what is required. There are only a handful of experts, but untold million amateurs. And the [contributions of these amateurs is absolutely essential](#) when you're trying to generate a website that contains a page for... well, everything. The world is a fractal place, filled with infinite detail. Nobody knows this better than software developers. The programmers in the trenches, spending every day struggling with the details, are the people who often have the most local knowledge about narrow programming topics. There just aren't enough experts to go around.

So what does it mean to be an expert, then, when **expertise is perceived as impractical at best, and a liability at worst**? In a [recent Google talk](#), James Bach presented the quintessential postmodern image of an expert performing—[Steve McQueen](#) in [The Towering Inferno](#):

[turns to fire commissioner] “What do we got here, Kappy?”

“Fire started, 81st floor, storage room. It's bad. Smoke's so thick, we can't tell how far it's spread.”

“Exhaust system?”

“Should've reversed automatically. It must be a motor burnout.”

“Sprinklers?”

“They're not working on 81.”

“Why not?”

“I don't know.”



[turns to architect] “Jim? Give us a quick refresher on your standpipe system.”

“Floors have 3 and 1.5 inch outlets.”

“GPM?”

“Fifteen hundred from ground to 68, and 1,000 from 68 to 100, and 500 from there to the roof.”

“Are these elevators programmed for emergencies?”

“Yes.”

“What floor are your plans on?”

“79. My office.”

“That's two floors below the fire. It'll be our Forward Command. Men, take up the equipment. I want to see all floor plans, 81 through 85.”

“Gotcha.”

[turns to security chief] “Give me a list of your tenants.”

“Don't worry, we're moving them out now.”

“Not live-ins. Businesses.”

“We lucked out. Most of them haven't moved in yet. Those that have are off at night.”

“I want to know who they are, not where.”

“What's that got to do with anything? Who they are?”

“Any wool or silk manufacturers? In a fire, wool and silk give off cyanide gas. Any sporting good manufacturers, like table-tennis balls? They give off toxic gases. Now do you want me to keep going?”

“One tenant list, coming up.”

[turns to crew leader] “What do we got?”

“Elevator bank, central core. Service elevators here. Air conditioning ducts, six inches.”

“Pipe alleys here?”

“One, two, three, four, five.”

“Have you got any construction on 81? Anything that can blow up, like gasoline, fabric cleaner?”

“I don't think so.”

What does this tell us? I mean, other than... Steve McQueen is a badass? **Being an expert isn't telling other people what you know.** It's understanding what questions to ask, and flexibly applying your knowledge to the specific situation at hand. Being an expert means providing sensible, highly contextual direction.

What I love about [James Bach's presentation](#) is how he spends the entire first half of it questioning and deconstructing everything—his field, his expertise, his own reputation and credentials, even! And then, only then, he cautiously, slowly builds it back up through a process of continual learning.

Level 0: I overcame *obliviousness*

I now realize there is something here to learn.

Level 1: I overcame *intimidation*

I feel I can learn this subject or skill. I know enough about it so that I am not intimidated by people who know more than me.

Level 2: I overcame *incoherence*

I no longer feel that I'm pretending or hand-waving. I feel reasonably competent to discuss or practice. What I say sounds like what I think I know.

Level 3: I overcame *competence*.

Now I feel productively self-critical, rather than complacently good enough. I want to take risks, invent, teach, and push myself. I want to be with other enthusiastic students.

Insight like this is why Mr. Bach is [my favorite Buccaneer-Scholar](#). He leaves us with this bit of advice to New Experts:

- Practice, practice, practice!
- Don't confuse experience with expertise.
- Don't trust folklore—but learn it anyway.
- Take nothing on faith. Own your methodology.
- Drive your own education—no one else will.
- Reputation = Money. Build and protect your reputation.
- Relentlessly gather resources, materials, and tools.
- Establish your standards and ethics.
- [Avoid certifications](#) that trivialize the craft.
- Associate with demanding colleagues.
- Write, speak, and *always tell the truth as you see it*.

Of course, Mr. Bach is talking about testing here, but I believe his advice applies equally well to developing expertise in programming, or anything else you might do in a professional capacity. It starts with questioning everything, most of all yourself.

So if you want to be an expert in practice rather than in name only, take a page from Steve McQueen's book. Don't be the guy telling everyone what to do. Be the guy asking all the questions.

OceanofPDF.com

On Our Project, We're Always 90 Percent Done

Although I love [reading programming books](#), I find software project management books to be some of the most mind-numbingly boring reading I've ever attempted. I suppose this means I probably shouldn't be a project manager. The bad news for the [Stack Overflow team](#) is that I effectively am one.

That's not to say that all software project management books are crap. Just most of them. One of the few that I've found compelling enough to finish is Johanna Rothman's "[Behind Closed Doors: Secrets of Great Management](#)." She co-wrote it with Esther Derby.

The
Pragmatic
Programmers

Behind Closed Doors



Secrets
of Great
Management

Johanna Rothman
Esther Derby

After reading it, you'll realize this is the book they should be handing out to every newly minted software project manager. And you'll be deeply depressed because you don't work with any software project managers who apparently have read it.

I originally discovered Johanna when one of her pieces was cited in the original Spolsky [Best Software Writing](#) book. Her article on [team compensation](#) basically blew my mind; it forced me to **rethink my entire perspective on being paid to work at a job**. You should read it. If you have a manager, you should get him or her to read it, too.

Since then, I've touched on her work briefly in “[Schedule Games](#)” and “[You Are Not Your Job](#).” But I'd like to focus on a specific aspect of project management that I'm apparently not very good at. A caller in [Podcast](#)

[#16](#) took me to task for [my original Stack Overflow schedule claims](#) way back in late April. What was supposed to be "six to eight weeks" became... well, something more like three months.

My problem is that I'm almost pathologically bad about writing things down. Unless I'm writing a blog entry, I suppose. I prefer to keep track of what I'm doing in my head, only anticipating as far ahead as the next item I plan to work on, while proceeding forward as quickly as I can. I think I fell prey, at least a little bit, to [this scenario](#):

'Look, Mike,' Tomas said. 'I can hand off my code today and call it 'feature complete', but I've probably got three weeks of cleanup work to do once I hand it off.' Mike asked what Tomas meant by 'cleanup.' 'I haven't gotten the company logo to show up on every page, and I haven't gotten the agent's name and phone number to print on the bottom of every page. It's little stuff like that. All of the important stuff works fine. I'm 99-percent done.'

Do you see the problem here? I know, there are so many it's [difficult to know where to begin listing them all](#), but what's the deepest, most fundamental problem at work here?

This software developer **does not have a detailed list of all the things he needs to do**. Which means, despite adamantly claiming that he is 99 percent done—he has no idea how long development will take! There's simply no factual basis for any of his schedule claims.

It is the job of a good software project manager to recognize the tell-tale symptoms of this classic mistake and address them head on before they derail the project. How? By forcing, encouraging developers to **create a detailed list of everything they need to do**. And then breaking that list down into sub-items. And then adding all the subitems they inevitably forgot because they didn't think that far ahead. Once you have all those items on a list, then—and only then—you can begin to estimate how long the work will take.

Until you've got at least the beginnings of a task list, any concept of scheduling is utter fantasy. A very pleasant fantasy, to be sure, but the real world can be extremely unforgiving to such dreams.

Johanna Rothman makes the same point in a recent email newsletter, and offers specific actions you can take to **avoid being stuck 90 percent done**:

1. List everything you need to do to finish the big chunk of work. I include any infrastructure work such as setting up branches in the source control system.
2. Estimate each item on that list. This initial estimate will help you see how long it might take to complete the entire task.
3. Now, look to see how long each item on that list will take to finish. If you have a task longer than one day, break that task into smaller pieces. Breaking larger tasks into these inch-pebbles is critical for escaping the 90 percent Done syndrome.
4. Determine a way to show visible status to anyone who's interested. If you're the person doing the work, what would you have to do to show your status to your manager? If you're the manager, what do you need to see? You might need to see lists of test cases or a demo or something else that shows you visible progress.
5. Since you've got one-day or smaller tasks, you can track your progress daily. I like to keep a chart or list of the tasks, my initial estimated end time and the actual end time for each task. This is especially important for you managers, so you can see if the person is being interrupted and therefore is multitasking. (See the article about the [Split Focus schedule game](#).)”

I'm not big on scheduling—or lists—but without the latter, I cannot have the former. It's like trying to defy the law of gravity. Thus, on our project, **we're always 90 percent done**. If you'd like to escape the 90 percent done ghetto on your software project, don't learn this the hard way, like I did. Every time someone asks you what your schedule is, you should be able to point to a list of everything you need to do. And if you can't—the first item on your task list should be to create that list.

Managing with Trust

[Marco Dorantes](#) recently linked to a great article by Watts Humphrey, who worked on IBM's OS/360 project: [Why Big Software Projects Fail](#). Watts opens with an analysis of software project completion data from 2001:

“Figure 2 shows another cut of the Standish data by project size. When looked at this way, half of the smallest projects succeeded, while none of the largest projects did. Since large projects still do not succeed even with all of the project management improvements of the last several years, one begins to wonder if large-scale software projects are inherently unmanageable.”

There's a strong correlation between project size and likelihood of failure. I'm sure that comes as no surprise; it's a lot easier to build a doghouse in your backyard than it is to build the [Brooklyn Bridge](#). What is surprising is the "radical" management solution he proposes for these large projects: **trust**.

“This question gets to the root of the problem with autocratic management methods: trust. If you trust and empower your software and other high-technology professionals to manage themselves, they will do extraordinary work. However, it cannot be blind trust. You must ensure that they know how to manage their own work, and you must monitor their work to ensure that they do it properly. The proper monitoring attitude is not to be distrustful, but instead, to show interest in their work. If you do not trust your people, you will not get their whole-hearted effort and you will not capitalize on the enormous creative potential of cohesive and motivated teamwork. It takes a leap of faith to trust your people, but the results are worth the risk.”

If you don't [delegate some measure of trust](#) to your teammates, can you even call it a team? Watts also notes that **trusting your team is not a substitute for managing them**. Trust shouldn't imply a free pass through the "how ya doin'?" school of feel-good non-management. That's what Paul Vick is complaining about in his [defense of the Microsoft Shipit award](#):

“As for the rest of [\[Joel Spolsky's\] article slagging the idea of performance reviews](#), I can only fall back on Churchill's immortal quote: ‘Democracy is the worst form of government except for all those others that have been tried.’ There's no question that performance reviews can have terrible effects, but what's the alternative? Give everyone a pat on the head, say ‘nice work’ and send them off to a nap with some warm milk and cookies? This isn't to say that there aren't better or worse ways to do performance reviews, but it seems cheap to dispatch them without suggesting some alternative.”

And he's right. In order to manage a project, you have to objectively measure what your teammates are doing—a delicate balancing act that DeMarco and Lister call [measuring with your eyes closed](#):

“In his 1982 book ‘Out of the Crisis,’ W. Edwards Deming set forth his now widely followed ‘Fourteen Points.’ Hidden among them, almost as an afterthought, is point 12B:

“Remove barriers that rob people in management and in engineering of their right to pride of workmanship. This means [among other things] abolishment of the annual or merit rating and of management by objectives.

“Even people who think of themselves as Deming-ites have trouble with this one. They are left gasping, What the hell are we supposed to do instead? Deming's point is that MBO and its ilk are managerial copouts. By using simplistic extrinsic motivators to goad performance, managers excuse themselves from harder matters such as investment, direct personal motivation, thoughtful team-formation, staff retention, and ongoing analysis and redesign of work procedures. Our point here is somewhat more limited: Any action that rewards team members differentially is likely to foster competition. Managers need to take steps to decrease or counteract this effect.

“Measuring with Your Eyes Closed: In order to make measurement deliver on its potential, management has to be perceptive and secure enough to cut itself out of the loop. Data collected on individual performance has to be used only to benefit that individual as an exercise in self-assessment. Only sanitized averages should be made available to the boss. If this is violated and the data is used for promotion or punitive action, the entire data

collection scheme will come to an abrupt halt. Individuals are inclined to do exactly what the manager would to improve themselves, so managers don't really need individual data in order to benefit from it.”

If this sounds difficult, well, that's because it is. Managing people is unbelievably difficult. Getting code to compile and pass all your unit tests? Piece of cake. Getting your team to work together? That's another matter entirely. Joel Spolsky [commented on Paul's post](#), elaborating on his position:

“The Shipit stupidity replaced a genuine form of employees being recognized for shipping a product (being given a copy of the shrinkwrapped box) with a ersatz form of recognition which made it pretty clear that management didn't even know that employees were already motivated for shipping software. And it's a classic case of gold-starism. It was universally derided by the hard core old-school developers that make Microsoft what it is today.”

Joel's problem with the Shipit awards was exactly the pitfall that DeMarco and Lister described. Managerial trust relationships take investment and work; **facile shortcuts like the Shipit award undermine this relationship**. Even if you're only building a doghouse, avoid taking these shortcuts.

OceanofPDF.com

Boyd's Law of Iteration

[Scott Stanfield](#) forwarded me a link to Roger Sessions' [A Better Path to Enterprise Architecture](#) yesterday. Even though it's got [the snake-oil word "Enterprise"](#) in the title, the article is surprisingly good.

I particularly liked the unusual analogy Roger chose to illustrate the difference between iterative and recursive approaches to software development. It starts with Air Force [Colonel John Boyd](#) researching a peculiar anomaly in the performance of 1950's era jet fighters:

“Colonel John Boyd was interested not just in any dogfights, but specifically in dogfights between [MiG-15s](#) and [F-86s](#). As an ex-pilot and accomplished aircraft designer, Boyd knew both planes very well. He knew the MiG-15 was a better aircraft than the F-86. The MiG-15 could climb faster than the F-86. The MiG-15 could turn faster than the F-86. The MiG-15 had better distance visibility.

“The F-86 had two points in its favor. First, it had better side visibility. While the MiG-15 pilot could see further in front, the F-86 pilot could see slightly more on the sides. Second, the F-86 had a hydraulic flight control. The MiG-15 had a manual flight control.

“The standing assumption on the part of airline designers was that maneuverability was the key component of winning dogfights. Clearly, the MiG-15, with its faster turning and climbing ability, could outmaneuver the F-86.

“There was just one problem with all this. Even though the MiG-15 was considered a superior aircraft by aircraft designers, the F-86 was favored by pilots. The reason it was favored was simple: in one-on-one dogfights with MiG-15s, the F-86 won nine times out of ten.”

How can an inferior aircraft consistently win over a superior aircraft? Boyd, who was himself one of the best dogfighters in history, had a theory:

“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*.

“The next question Boyd asked is this: why would the F-86 iterate faster? The reason, he concluded, was something that nobody had thought was particularly important. It was the fact that the F-86 had a hydraulic flight stick whereas the MiG-15 had a manual flight stick.



“Without hydraulics, it took slightly more physical energy to move the MiG-15 flight stick than it did the F-85 flight stick. Even though the MiG-15 would turn faster (or climb higher) once the stick was moved, the amount of energy it took to move the stick was greater for the MiG-15 pilot.

“With each iteration, the MiG-15 pilot grew a little more fatigued than the F-86 pilot. And as he gets more fatigued, it took just a little bit longer to complete his OOPA loop. The MiG-15 pilot didn't lose because he got outfought. He lost because he got out-OOPAed.”

This leads to **Boyd's Law of Iteration: speed of iteration beats quality of iteration.**

You'll find this same theme echoed throughout every discipline of modern software engineering:

- Unit tests should be [small and fast](#), so you can run them with every build.
- Usability tests work best if you [make small changes every two weeks and quickly discard what isn't working](#).
- Most agile approaches recommend iterations [no longer than 4 weeks](#).
- Software testing is about [failing early and often](#).
- Functional specifications are best when they're [concise and evolving](#).

When in doubt, iterate faster.

OceanofPDF.com

Overnight Success: It Takes Years

Paul Buchheit, the original lead developer of Gmail, notes that [the success of Gmail was a long time in coming](#):

“We starting working on Gmail in August 2001. For a long time, almost everyone disliked it. Some people used it anyway because of the search, but they had endless complaints. Quite a few people thought that we should kill the project, or perhaps ‘reboot’ it as an enterprise product with native client software, not this crazy Javascript stuff. Even when we got to the point of launching it on April 1, 2004—two and a half years after starting work on it—many people inside of Google were predicting doom. The product was too weird, and nobody wants to change email services. I was told that we would never get a million users.

“Once we launched, the response was surprisingly positive, except from the people who hated it for a variety of reasons. Nevertheless, it was frequently described as ‘niche,’ and ‘not used by real people outside of silicon valley.’

“Now, almost seven and a half years after we started working on Gmail, I see [an [article](#) describing how Gmail grew 40 percent last year, compared to two percent for Yahoo and negative seven percent for Hotmail].”

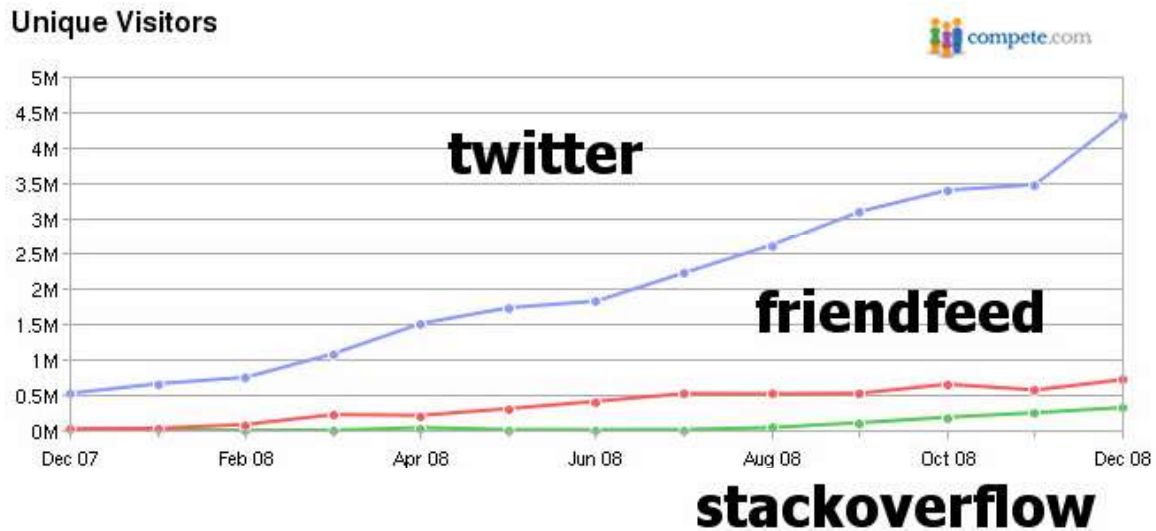
Paul has since left Google and now works at his own startup, [FriendFeed](#). Many industry insiders have not been kind to FriendFeed. Stowe Boyd even went so far as to [call FriendFeed a failure](#). Paul takes this criticism in stride:

“Creating an important new product generally takes time. FriendFeed needs to continue changing and improving, just as Gmail did six years ago. FriendFeed shows a lot of promise, but it's still a ‘work in progress.’

“My expectation is that big success takes years, and there aren't many counter-examples (other than YouTube, and they didn't actually get to the point of making piles of money just yet). Facebook grew very fast, but it's almost five years old at this point. Larry and Sergey started working on Google in 1996—when I started there in 1999, few people had heard of it yet.

“This notion of overnight success is very misleading, and rather harmful. If you're starting something new, expect a long journey. That's no excuse to move slow though. To the contrary, you must move very fast, otherwise you will never arrive, because it's a long journey! This is also why it's important to be frugal—you don't want to [starve to death halfway up the mountain](#).”

Stowe Boyd illustrated [his point about FriendFeed](#) with a graph comparing Twitter and FriendFeed traffic. Allow me to update Mr. Boyd's graph with another data point of my own.



I find Paul's attitude refreshing, because I take the same attitude toward our startup, [Stack Overflow](#). I have zero expectation or even desire for overnight success. What I am planning is several years of grinding through constant, steady improvement.

This business plan isn't much different from my career development plan: success takes years. And when I say years, I really mean it! Not as some cliched regurgitation of "work smarter, not harder." I'm talking actual calendar years. You know, of the 12 months, 365 days variety. You will literally have to spend multiple years of your life grinding away at this stuff, waking up every day and doing it over and over, practicing and gathering feedback each day to continually get better. It might be unpleasant at times and even downright un-fun occasionally, but it's necessary.

This is hardly unique or interesting advice. Peter Norvig's classic [Teach Yourself Programming in Ten Years](#) already covered this topic far better

than I.

“Researchers have shown it takes about ten years to develop expertise in any of a wide variety of areas, including chess playing, music composition, telegraph operation, painting, piano playing, swimming, tennis, and research in neuropsychology and topology. The key is *deliberative* practice: not just doing it again and again, but challenging yourself with a task that is just beyond your current ability, trying it, analyzing your performance while and after doing it, and correcting any mistakes. Then repeat. And repeat again.

“There appear to be no real shortcuts: even Mozart, who was a musical prodigy at age four, took 13 more years before he began to produce world-class music. The Beatles seemed to burst onto the scene with a string of number one hits and an appearance on the Ed Sullivan show in 1964. But they had been playing small clubs in Liverpool and Hamburg since 1957, and while they had mass appeal early on, their first great critical success, ‘Sgt. Peppers,’ was released in 1967.”

Honestly, I look forward to waking up someday two or three years from now and doing the exact same thing I did today: working on the Stack Overflow code, eking out yet another tiny improvement or useful feature. Obviously we want to succeed. But on some level, success is irrelevant, because the process is inherently satisfying. Waking up every day and doing something you love—even better, surrounded by a community who loves it too—is its own reward. Despite being a metric ton of work.

The blog is no different. I often give aspiring bloggers [this key piece of advice](#): if you're starting a blog, don't expect anyone to read it for six months. If you do, I can guarantee you will be sorely disappointed. However, if you can stick to a posting schedule and produce one or two quality posts every week for an entire calendar year... then, and only then, can you expect to see a trickle of readership. I started this blog in 2004, and it took a solid three years of writing three to five times per week before it [achieved](#) anything resembling popularity within the software development community.

I fully expect to be writing on this blog, in one form or another, for the rest of my life. It is a part of who I am. And with that bit of drama out of the

way, I have no illusions: ultimately, I'm [just the guy on the internet who writes that blog](#).



That's perfectly fine by me. I never said I was clever.

Whether you ultimately achieve readers, or pageviews, or whatever [high score table](#) it is we're measuring this week, try to remember it's worth doing because, well—it's worth doing.

And if you keep doing it long enough, who knows? You might very well wake up one day and find out you're an overnight success.

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

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

OceanofPDF.com

II.

Programming

OceanofPDF.com

How to Become a Better Programmer by Not Programming

Last year in [Programmers as Human Beings](#), I mentioned that I was reading [Programmers At Work](#). It's a great collection of interviews with famous programmers circa 1986. All the interviews are worth reading, but the interview with Bill Gates has one particular answer that cuts to the bone:

Does accumulating experience through the years necessarily make programming easier?

Bill Gates: “No. I think after the first three or four years, it's pretty cast in concrete whether you're a good programmer or not. After a few more years, you may know more about managing large projects and personalities, but after three or four years, it's clear what you're going to be. There's no one at Microsoft who was just kind of mediocre for a couple of years, and then just out of the blue started optimizing everything in sight. I can talk to somebody about a program that he's written and know right away whether he's really a good programmer.”

We already know [there's a vast divide between those who can program and those who cannot](#).

But the dirty little secret of the software development industry is that this is also true even for people who can program: [there's a vast divide between good developers and mediocre developers](#). A mediocre developer can program his or her heart out for four years, but that won't magically transform them into a good developer. And the good developers always seem to have a natural knack for the stuff from the very beginning.

I agree with Bill. From what I've seen, there's just no crossing the skill chasm as a software developer. You've either got it, or you don't. No amount of putting your nose to the grindstone will change that. But if you accept that premise, it also presents us with a paradox: if experience doesn't make you a better programmer, what does? Are our skill levels written in stone? Is it impossible to become a better programmer?

To answer that question, you have to consider the obsessive nature of programming itself. Good developers are good at programming. Really good at programming. You might even say fanatically good. If they're anything like me, they've spent nearly every waking moment in front of a computer for most of their lives. And naturally, they get better at it over time. Competent software developers have already mastered the skill of programming, which puts them in a very select club. But if you're already in the 97th percentile for programming aptitude, what difference does a few more percentile points really make in the big scheme of things?

The older I get, the more I believe that **the only way to become a better programmer is by not programming**. You have to come up for air, put down the compiler for a moment, and take stock of what you're really doing. Code is important, but it's [a small part of the overall process](#).

This [piece in Design Observer](#) offers a nice bit of related advice:

“Over the years, I came to realize that my best work has always involved subjects that interested me, or—even better—subjects about which I've become interested, and even passionate about, through the very process of doing design work. I believe I'm still passionate about graphic design. But the great thing about graphic design is that it is almost always about something else. Corporate law. Professional football. Art. Politics. Robert Wilson. And if I can't get excited about whatever that something else is, I really have trouble doing a good work as a designer. To me, the conclusion is inescapable: the more things you're interested in, the better your work will be.”

Passion for coding is a wonderful thing. But it's all too easy to mindlessly, reflexively entrench yourself deeper and deeper into a skill that you've already proven yourself more than capable at many times over. To truly become a better programmer, you have to to **cultivate passion for everything else that goes on around the programming**.

Bill Gates, in a 2005 interview, [follows up in spirit to his 1986 remarks](#):

“The nature of these jobs is not just closing your door and doing coding, and it's easy to get that fact out. The greatest missing skill is somebody who's both good at understanding the engineering and who has good relationships with the hard-core engineers, and bridges that to working with

the customers and the marketing and things like that. And so that sort of engineering management career track, even amongst all the people we have, we still fall short of finding people who want to do that, and so we often have to push people into it.

“I'd love to have people who come to these jobs wanting to think of it as an exercise in people management and people dynamics, as well as the basic engineering skills. That would be absolutely amazing.

“And we can promise those people within two years of starting that career most of what they're doing won't be coding, because there are many career paths, say, within that Microsoft Office group where you're part of creating this amazing product, you get to see how people use it, you get to then spend two years, build another version, and really change the productivity in this very deep way, take some big bets on what you're doing and do some things that are just responsive to what that customer wants.”

You won't—you cannot—become a better programmer through sheer force of programming alone. You can only complement and enhance your existing programming skills by branching out. Learn about your users. Learn about the industry. Learn about your business.

The more things you are interested in, the better your work will be.

OceanofPDF.com

The Broken Window Theory

In a [previous entry](#), I touched on the broken window theory. You might be familiar with [the Pragmatic Programmers' take on this](#):

“Don't leave "broken windows" (bad designs, wrong decisions, or poor code) unrepaired. Fix each one as soon as it is discovered. If there is insufficient time to fix it properly, then board it up. Perhaps you can comment out the offending code, or display a "Not Implemented" message, or substitute dummy data instead. Take some action to prevent further damage and to show that you're on top of the situation.

“We've seen clean, functional systems deteriorate pretty quickly once windows start breaking. There are other factors that can contribute to software rot, and we'll touch on some of them elsewhere, but neglect accelerates the rot faster than any other factor.”

That's excellent advice for programmers, but it's not the complete story.



The broken window theory is based on an [Atlantic Monthly article](#) published in 1982. It's worth reading the article to get a deeper understanding of the human factors driving the theory:

“Second, at the community level, disorder and crime are usually inextricably linked, in a kind of developmental sequence. Social psychologists and police officers tend to agree that if a window in a building is broken and is left unrepaired, all the rest of the windows will soon be broken. This is as true in nice neighborhoods as in rundown ones. Window-breaking does not necessarily occur on a large scale because some areas are inhabited by determined window-breakers whereas others are populated by window-lovers; rather, one unrepaired broken window is a signal that no one cares, and so breaking more windows costs nothing. (It has always been fun.)

“Philip Zimbardo, a Stanford psychologist, reported in 1969 on some experiments testing the broken-window theory. He arranged to have an automobile without license plates parked with its hood up on a street in the Bronx and a comparable automobile on a street in Palo Alto, California. The car in the Bronx was attacked by ‘vandals’ within ten minutes of its ‘abandonment.’ The first to arrive were a family—father, mother, and young son—who removed the radiator and battery. Within twenty-four hours, virtually everything of value had been removed. Then random destruction began—windows were smashed, parts torn off, upholstery ripped. Children began to use the car as a playground. Most of the adult “vandals” were well-dressed, apparently clean-cut whites. The car in Palo Alto sat untouched for more than a week. Then Zimbardo smashed part of it with a sledgehammer. Soon, passersby were joining in. Within a few hours, the car had been turned upside down and utterly destroyed. Again, the ‘vandals’ appeared to be primarily respectable whites.

“Untended property becomes fair game for people out for fun or plunder and even for people who ordinarily would not dream of doing such things and who probably consider themselves law-abiding. Because of the nature of community life in the Bronx—its anonymity, the frequency with which cars are abandoned and things are stolen or broken, the past experience of ‘no one caring’—vandalism begins much more quickly than it does in staid Palo Alto, where people have come to believe that private possessions are

cared for, and that mischievous behavior is costly. But vandalism can occur anywhere once communal barriers—the sense of mutual regard and the obligations of civility—are lowered by actions that seem to signal that ‘no one cares.’”

There's even [an entire book on this subject](#). What's fascinating to me is that the mere perception of disorder—even with seemingly irrelevant petty crimes like graffiti or minor vandalism—precipitates a negative feedback loop that can result in total disorder:

“We suggest that "untended" behavior also leads to the breakdown of community controls. A stable neighborhood of families who care for their homes, mind each other's children, and confidently frown on unwanted intruders can change, in a few years or even a few months, to an inhospitable and frightening jungle. A piece of property is abandoned, weeds grow up, a window is smashed. Adults stop scolding rowdy children; the children, emboldened, become more rowdy. Families move out, unattached adults move in. Teenagers gather in front of the corner store. The merchant asks them to move; they refuse. Fights occur. Litter accumulates. People start drinking in front of the grocery; in time, an inebriate slumps to the sidewalk and is allowed to sleep it off. Pedestrians are approached by panhandlers.

“At this point it is not inevitable that serious crime will flourish or violent attacks on strangers will occur. But many residents will think that crime, especially violent crime, is on the rise, and they will modify their behavior accordingly. They will use the streets less often, and when on the streets will stay apart from their fellows, moving with averted eyes, silent lips, and hurried steps. ‘Don't get involved.’ For some residents, this growing atomization will matter little, because the neighborhood is not their ‘home’ but ‘the place where they live.’ Their interests are elsewhere; they are cosmopolitans. But it will matter greatly to other people, whose lives derive meaning and satisfaction from local attachments rather than worldly involvement; for them, the neighborhood will cease to exist except for a few reliable friends whom they arrange to meet.”

Programming is insanely detail oriented, and perhaps this is why: if you're not on top of the details, the perception is that things are out of control, and

it's only a matter of time before your project spins out of control. **Maybe we should be sweating the small stuff.**

OceanofPDF.com

Programming: Love It or Leave It

In a recent Joel on Software forum post [Thinking of Leaving the Industry](#), one programmer wonders if software development is the right career choice in the face of broad economic uncertainty:

“After reading the disgruntled posts here from long time programmers and hearing so much about ageism and outsourcing, I'm thinking of leaving the industry. What is a good industry to get into where your programming skills would put you at an advantage?”

Joel Spolsky responded:

“Although the tech industry is not immune, programming jobs are not really being impacted. Yes, there are fewer openings, but there are still openings (see my job board for evidence). I still haven't met a great programmer who doesn't have a job. I still can't fill all the openings at my company.

“Our pay is great. There's no other career except Wall Street that regularly pays kids \$75,000 right out of school, and where so many people make six figures salaries for long careers with just a bachelors degree. There's no other career where you come to work every day and get to invent, design, and engineer the way the future will work.

“Despite the occasional idiot bosses and workplaces that forbid you from putting up Dilbert cartoons on your cubicle walls, there's no other industry where workers are treated so well. Jesus you're spoiled, people. Do you know how many people in America go to jobs where you need permission to go to the bathroom?”

“Stop the whining, already. Programming is a fantastic career. Most programmers would love to do it even if they didn't get paid. How many people get to do what they love and get paid for it? 2 percent? 5 percent?”

I tend to agree with Joel's brand of tough love. What he seems to be saying—after taking my usual poetic license—is this:

Programming: love it or leave it.

Unless you're fortunate enough to work for a top tier software development company, like Google, Microsoft, or Apple, you've probably experienced first hand the [huge skill disparities in your fellow programmers](#). I'm betting you've also wondered more than once [why some of your coworkers can't, well, program](#). Even if that's what their job description says.

Over the last twenty years, I've worked with far too many programmers who honestly **had no business being paid to be a programmer**. Now, I'm not talking about your average programmer here. We're all human, and we all make mistakes. I'm talking about [the Daily WTF crew](#). People that actively give programming a bad name, and you, as their coworker, a constant headache.

Like Joel, I'm not ready to call the current conditions [a new dot com bubble](#) yet, because business is still quite good. But one of the (very) few bright spots of the previous bubble was that **it weeded out all the people who didn't truly love software development**. Once the incentive to become an overnight dot-com genius programmer millionaire was gone, computer science enrollment suddenly dropped precipitously at colleges across the country. The only people left applying for programming jobs were the true freaks and geeks who, y'know, [loved this stuff](#). The kind of people I had originally enjoyed working with so much. At least until a bunch of careerist gold diggers suddenly showed up and started polluting our workplace.

As much as the dot com bubble sucked, I was intensely glad to see these people go. Now I'm wondering if the current economic conditions are an opportunity to clean house again.

I mean this in the nicest possible way, but not everyone should be a programmer. How often have you wished that a certain coworker of yours would suddenly have an epiphany one day and decide that this whole software engineering thing just isn't working out for them? How do you tell someone that the quality of their work is terrible and [they'll never be good at their job](#)—so much so that they should literally quit and pursue a new career? I've wanted to many times, but I never had the guts.

Joel implied that good programmers love programming so much they'd do it for no pay at all. I won't go quite that far, but I will note that the best

programmers I've known have all had a **lifelong passion for what they do**. There's no way a minor economic blip would ever convince them they should do anything else. No way. No how.

So if a programmer ever hints, even in passing, that they might possibly want to exit the field—they probably should. I'm not saying you should be a jerk about it, obviously. But if someone has any doubt at all about programming as a career choice, they should be encouraged to explore alternatives—and make room for another programmer who [unashamedly loves to code](#).

Then again, maybe I'm not the best person to ask. I spent Christmas Eve [setting up servers](#). I'm on holiday right now, sitting in a hotel room in Santa Barbara, and you know what I spent the last two nights doing until the wee hours of the morning? Writing code to improve [Stack Overflow](#). Oh yeah, and this blog post.

So I might be a little biased.

OceanofPDF.com

Some Lessons from Forth

It's easy to get caught up in the "newer is better" mindset of software development and forget that [ideas are more important than code](#). Not everything we do is obsolete in four years. The [Evolution of Forth](#), which outlines Charles Moore's guiding principles in creating and implementing the [FORTH language](#), is an excellent illustration of the timelessness of ancient computer wisdom:

1. **Keep it simple:** As the number of capabilities you add to a program increases, the complexity of the program increases exponentially. The problem of maintaining compatibility among these capabilities, to say nothing of some sort of internal consistency in the program, can easily get out of hand. You can avoid this if you apply the Basic Principle. You may be acquainted with an operating system that ignored the Basic Principle. It is very hard to apply. All the pressures, internal and external, conspire to add features to your program. After all, it only takes a half-dozen instructions, so why not? The only opposing pressure is the Basic Principle, and if you ignore it, there is no opposing pressure.
2. **Do not speculate:** Do not put code in your program that might be used. Do not leave hooks on which you can hang extensions. The things you might want to do are infinite; that means that each has 0 probability of realization. If you need an extension later, you can code it later—and probably do a better job than if you did it now. And if someone else adds the extension, will he notice the hooks you left? Will you document this aspect of your program?
3. **Do it yourself:** The conventional approach, enforced to a greater or lesser extent, is that you shall use a standard subroutine. I say that you should write your own subroutines. Before you can write your own subroutines, you have to know how. This means, to be practical, that you have written it before; which makes it difficult to get started. But

give it a try. After writing the same subroutine a dozen times on as many computers and languages, you'll be pretty good at it.

I covered the first two points before in [KISS and YAGNI](#). Point 3 is more subtle. It seems to fly in the face of [don't repeat yourself](#), but what he's really saying—and I agree—is that **you have to make your own mistakes to truly learn**. There's a world of difference between someone explaining "you should always index your tables because it's a best practice," and having your app get progressively slower as records are added to the table. (You laugh, but I've worked with developers who did this.) You learn "why" a lot faster when you're actually experiencing it instead of passively reading about it.

Moore characterizes **simplicity as a force that must be applied** instead of a passive goal. And he's right—all too often, I see developers failing to make the hard choices necessary to keep their applications simple. It's easier to [just say yes](#) to everything.

OceanofPDF.com

The Joy of Deletion

I generally dislike these kinds of "Me, too!" posts, but I have to make an exception for Ned Batchelder's [excellent blog entry on deleting code](#). I've often run into this phenomenon with other developers, and it bugged the heck out of me, although I couldn't quantify exactly why. Well, now I can:

“If you have a chunk of code you don't need any more, there's one big reason to delete it for real rather than leaving it in a disabled state: to reduce noise and uncertainty. Some of the worst enemies a developer has are noise or uncertainty in his code, because they prevent him from working with it effectively in the future.

“A chunk of code in a disabled state just causes uncertainty. It puts questions in other developers' minds:

- Why did the code used to be this way?
- Why is this new way better?
- Are we going to switch back to the old way?
- How will we decide?

“If the answer to one of these questions is important for people to know, then write a comment spelling it out. Don't leave your co-workers guessing.”

I have been angrily accused of deleting someone's commented code on more than one occasion. I say, give me a reason not to delete it, and I won't. Otherwise, it's fair game. In my experience this kind of "oh, I'll get back to it" code just sits in the codebase forever, junking up the works for every future developer.

Separating Programming Sheep from Non-Programming Goats

A bunch of people have linked to this [academic paper](#), which proposes a **way to separate programming sheep from non-programming goats in computer science classes**—long before the students have ever touched a program or a programming language:

“All teachers of programming find that their results display a 'double hump'. It is as if there are two populations: those who can [program], and those who cannot [program], each with its own independent bell curve. Almost all research into programming teaching and learning have concentrated on teaching: change the language, change the application area, use an IDE and work on motivation. None of it works, and the double hump persists. We have a test which picks out the population that can program, before the course begins. We can pick apart the double hump. You probably don't believe this, but you will after you hear the talk. We don't know exactly how/why it works, but we have some good theories.”

I wasn't aware that the dichotomy between programmers and non-programmers was so pronounced at this early stage. Dan Bricklin touched on this topic in his essay, “[Why Johnny Can't Program](#).” But evidently it's common knowledge amongst those who teach computer science:

“Despite the enormous changes which have taken place since electronic computing was invented in the 1950s, some things remain stubbornly the same. In particular, most people can't learn to program: between 30 percent and 60 percent of every university computer science department's intake fail the first programming course. Experienced teachers are weary but never oblivious of this fact; brighteyed beginners who believe that the old ones must have been doing it wrong learn the truth from bitter experience; and so it has been for almost two generations, ever since the subject began in the 1960s.”

You may think the test they're proposing to determine programming aptitude is complex, but it's not. Here's question one, verbatim:

Read the following statements and tick the box next to the correct answer.

int a = 10;int b = 20;a = b;The new values of a and b are:
[] a = 20 b = 0 [] a = 20 b = 20
[] a = 0 b = 10 [] a = 10 b = 10 [] a = 30 b = 20
[] a = 30 b = 0 [] a = 10 b = 30 [] a = 0 b = 30
[] a = 10 b = 20 [] a = 20 b = 10

This test seems trivial to professional programmers, but remember, it's intended for students who have never looked at a line of code in their lives. The other 12 questions are all variations on the same assignment theme.

The authors of the paper posit that the primary hurdles in computer science are..

1. assignment and sequence
2. recursion / iteration
3. concurrency*

... in that order. Thus, we start by testing the very first hurdle novice programmers will encounter: assignment. The test results divided the students cleanly into three groups:

- 44 percent of students formed a consistent mental model of how assignment works (even if incorrect!)
- 39 percent students never formed a consistent model of how assignment works.
- 8 percent of students didn't give a damn and left the answers blank.

The test was administered twice; once at the beginning, before any instruction at all, and again after three weeks of class. The striking thing is that there was virtually no movement at all between the groups from the first to second test. Either you had a consistent model in your mind

immediately upon first exposure to assignment, the first hurdle in programming—or else you never developed one!

The authors found an extremely high level of correlation between success at programming and forming a consistent mental model:

“Clearly, Dehnahti's test is not a perfect divider of programming sheep from non-programming goats. Nevertheless, if it were used as an admissions barrier, and only those who scored consistently were admitted, the pass/fail statistics would be transformed. In the total population 32 out of 61 (52 percent) failed; in the first-test consistent group only 6 out of 27 (22 percent). We believe that we can claim that we have a predictive test which can be taken prior to the course to determine, with a very high degree of accuracy, which students will be successful. This is, so far as we are aware, the first test to be able to claim any degree of predictive success.”

I highly recommend reading through [the draft paper](#), which was remarkably entertaining for what I thought was going to be a dry, academic paper. Instead, it reads like a blog entry. It's filled with interesting insights like this one:

“It has taken us some time to dare to believe in our own results. It now seems to us, although we are aware that at this point we do not have sufficient data, and so it must remain a speculation, that what distinguishes the three groups in the first test is their different attitudes to meaninglessness.

“Formal logical proofs, and therefore programs formal logical proofs that particular computations are possible, expressed in a formal system called a programming language are utterly meaningless. To write a computer program you have to come to terms with this, to accept that whatever you might want the program to mean, the machine will blindly follow its meaningless rules and come to some meaningless conclusion. In the test the consistent group showed a pre-acceptance of this fact: they are capable of seeing mathematical calculation problems in terms of rules, and can follow those rules wheresoever they may lead. The inconsistent group, on the other hand, looks for meaning where it is not. The blank group knows that it is looking at meaninglessness, and refuses to deal with it.”

Everyone should know how to use a computer, but not everyone needs to be a programmer. But it's still a little disturbing that **the act of programming seems literally unteachable to a sizable subset of incoming computer science students**. Evidently not everyone is as fascinated by meaningless rules and meaningless conclusions as we are; I can't imagine why not.

OceanofPDF.com

Are You Following the Instructions on the Paint Can?

We're currently undertaking some painting projects at home. Which means I'll be [following the instructions on the paint can](#).



But what would happen if I didn't follow the instructions on the paint can? Here's a list of common interior painting mistakes:

The single most common mistake in any project is failure to read and follow manufacturer's instructions for tools and materials being used. In regard to painting, the most common mistakes are:

- Not preparing a clean, sanded, and primed (if needed) surface.

- Failure to mix the paints properly.
- Applying too much paint to the applicator.
- Using water-logged applicators.
- Not solving dampness problems in the walls or ceilings.
- Not roughing up enamel paint before painting over it.

What I find particularly interesting is that **none of the mistakes on this checklist have anything to do with my skill as a painter.** My technical proficiency (or lack thereof) as a painter doesn't even register! To guarantee a reasonable level of quality, you don't have to spend weeks practicing your painting skills. You don't even have to be a good painter. All you have to do is follow the instructions on the paint can!

Sure, it seems like common sense. But take a close look at the houses on the streets you drive by. Each street has that one house where the owners, for whatever reason, chose not to follow the instructions on the paint can.

For years, software development was an entire subdivision of badly painted houses. But the field of software development is now mature enough that we have a number of [paint cans](#) to [refer to](#). Here's [one such checklist from Joel Spolsky](#), circa 2000:

1. Do you use source control?
2. Can you make a build in one step?
3. Do you make daily builds?
4. Do you have a bug database?
5. Do you fix bugs before writing new code?
6. Do you have an up-to-date schedule?
7. Do you have a spec?
8. Do programmers have quiet working conditions?
9. Do you use the best tools money can buy?
10. Do you have testers?

11. Do new candidates write code during their interview?

12. Do you do hallway usability testing?

The type of paint can you choose—and the instructions you follow—are highly debatable, of course. But make sure, at the very least, you're following the instructions on the paint can for your software development project.

OceanofPDF.com

Curly's Law: Do One Thing

In [Outliving the Great Variable Shortage](#), Tim Ottinger invokes Curly's Law:

“A variable should mean one thing, and one thing only. It should not mean one thing in one circumstance, and carry a different value from a different domain some other time. It should not mean two things at once. It must not be both a floor polish and a dessert topping. It should mean One Thing, and should mean it all of the time.”

The late, great Jack Palance played grizzled cowboy Curly Washburn in the 1991 comedy “[City Slickers](#).” Curly's Law is defined in this bit of dialog from the movie:



Curly: Do you know what the secret of life is?

Curly: This. [holds up one finger]

Mitch: Your finger?

Curly: One thing. Just one thing. You stick to that and the rest don't mean shit.

Mitch: But what is the "one thing?"

Curly: [smiles] That's what *you* have to find out.

Curly's Law, Do One Thing, is reflected in several core principles of modern software development:

[Don't Repeat Yourself](#)

If you have more than one way to express the same thing, at some point the two or three different representations will most likely fall out of step with each other. Even if they don't, you're guaranteeing yourself the headache of maintaining them in parallel whenever a change occurs. And change will occur. Don't repeat yourself is important if you want flexible and maintainable software.

[Once and Only Once](#)

Each and every declaration of behavior should occur once, and only once. This is one of the main goals, if not the main goal, when refactoring code. The design goal is to eliminate duplicated declarations of behavior, typically by merging them or replacing multiple similar implementations with a unifying abstraction.

[Single Point of Truth](#)

Repetition leads to inconsistency and code that is subtly broken, because you changed only some repetitions when you needed to change all of them. Often, it also means that you haven't properly thought through the organization of your code. Any time you see duplicate code, that's a danger sign. Complexity is a cost; don't pay it twice.

Although Curly's Law definitely applies to normalization and removing redundancies, Do One Thing is more nuanced than the various restatements of Do Each Thing Once outlined above. It runs deeper. Bob Martin refers to it as [The Single Responsibility Principle](#):

The Single Responsibility Principle says that a class should have one, and only one, reason to change. As an example, imagine the following class:

```
class Employee{ public Money calculatePay() public void save() public String reportHours() }
```

This class violates the SRP because it has three reasons to change:

1. *The business rules having to do with calculating pay.*
2. *The database schema.*
3. *The format of the string that reports hours.*

We don't want a single class to be impacted by these three completely different forces. We don't want to modify the Employee class every time the accounts decide to change the format of the hourly report, or every time the DBAs make a change to the database schema, as well as every time the managers change the payroll calculation. Rather, we want to separate these functions out into different classes so that they can change independently of each other.

Curly's Law is about choosing a single, clearly defined goal for any particular bit of code: Do One Thing. That much is clear. But in choosing one thing, you are ruling out an infinite universe of other possible things you could have done. **Curly's Law also means consciously choosing what your code won't do.** This is much more difficult than choosing what to do, because it runs counter to all the natural generalist tendencies of software developers. It could mean breaking code apart, violating traditional OOP rules, or introducing duplicate code. It's taking one step backward to go two steps forward.

Each variable, each line of code, each function, each class, each project should Do One Thing. Unfortunately, we usually don't find out what that one thing is until we've [reached the end of it](#).

OceanofPDF.com

The Ultimate Code Kata

As I was paging through Steve Yegge's [voluminous body](#) of work recently, I was struck by a 2005 entry on [practicing programming](#):

“Contrary to what you might believe, merely doing your job every day doesn't qualify as real practice. Going to meetings isn't practicing your people skills, and replying to mail isn't practicing your typing. You have to set aside some time once in a while and do focused practice in order to get better at something.

“I know a lot of great engineers—that's one of the best perks of working at Amazon—and if you watch them closely, you'll see that they practice constantly. As good as they are, they still practice. They have all sorts of ways of doing it, and this essay will cover a few of them.

“The great engineers I know are as good as they are *because* they practice all the time. People in great physical shape only get that way by working out regularly, and they need to keep it up, or they get out of shape. The same goes for programming and engineering.”

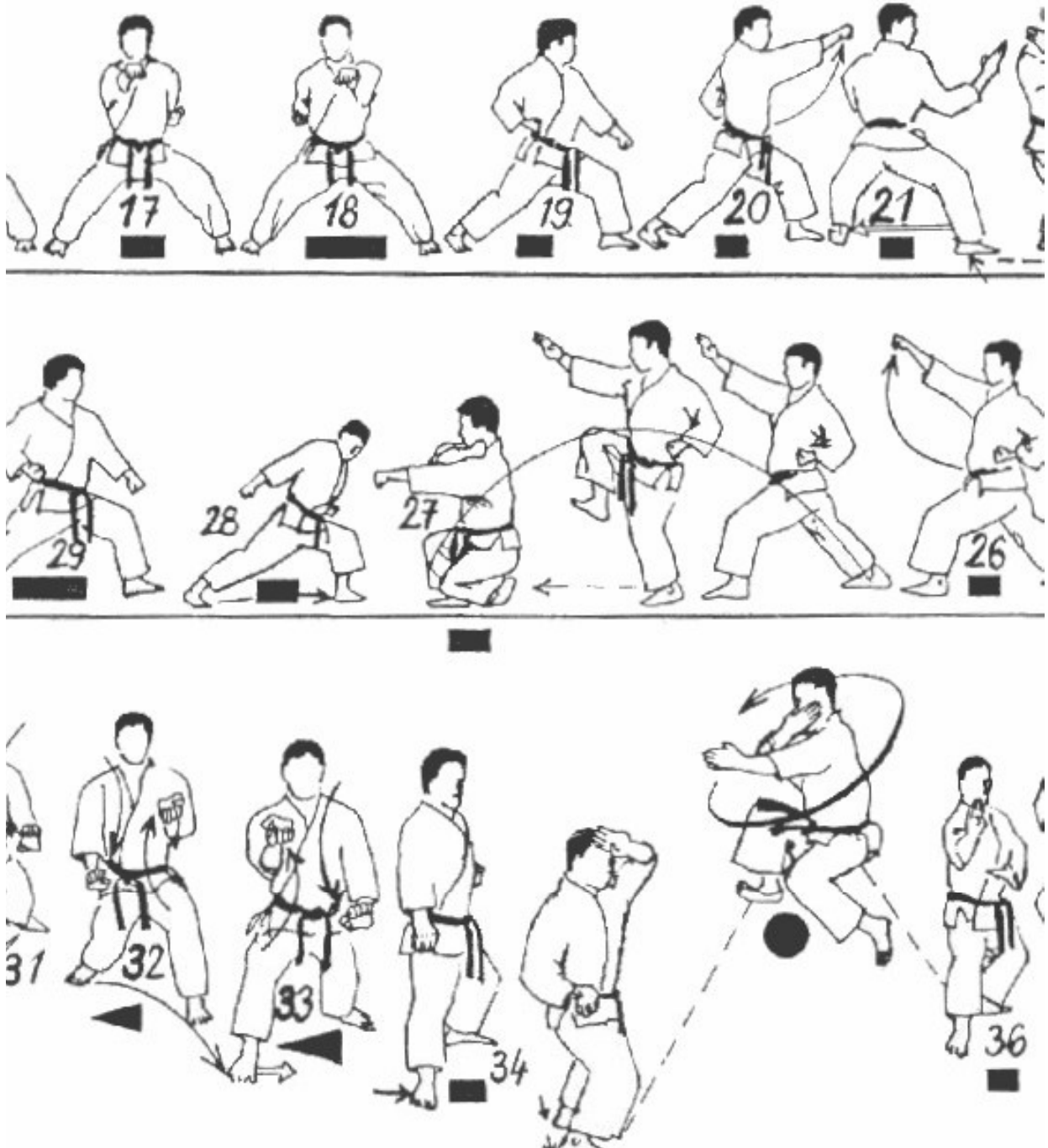
It's an important distinction. I may drive to work every day, but I'm far from a professional driver. Similarly, programming every day may not be enough to make you a professional programmer. So what can turn someone into a professional driver or programmer? What do you do to practice?

The answer lies in the Scientific American article [The Expert Mind](#):

“Ericsson argues that what matters is not experience per se but ‘effortful study,’ which entails continually tackling challenges that lie just beyond one's competence. That is why it is possible for enthusiasts to spend tens of thousands of hours playing chess or golf or a musical instrument without ever advancing beyond the amateur level and why a properly trained student can overtake them in a relatively short time. It is interesting to note that time spent playing chess, even in tournaments, appears to contribute less than such study to a player's progress; the main training value of such games is to point up weaknesses for future study.”

Effortful study means constantly tackling problems at the very edge of your ability. Stuff you may have a high probability of failing at. Unless you're [failing some of the time](#), you're probably not growing professionally. You have to seek out those challenges and push yourself beyond your comfort limit.

Those challenges can sometimes be found on the job, but they don't have to be. Separating the practicing from the profession is often referred to as [code kata](#).



The concept of kata, a series of choreographed practice movements, is [borrowed from the martial arts](#).

If you're looking for some examples of code kata—ways to practice effortful study and hone your programming skills—[Steve's article](#) has some excellent starting points. He calls them **practice drills**:

1. Write your resume. List all your relevant skills, then note the ones that will still be needed in 100 years. Give yourself a 1-10 rating in each skill.
2. Make a list of programmers who you admire. Try to include some you work with, since you'll be borrowing them for some drills. Make one or two notes about things they seem to do well—things you wish you were better at.
3. Go to Wikipedia's [entry for computer science](#), scroll down to the "Prominent pioneers in computer science" section, pick a person from the list, and read about them. Follow any links from there that you think look interesting.
4. Read through someone else's code for 20 minutes. For this drill, alternate between reading great code and reading bad code; they're both instructive. If you're not sure of the difference, ask a programmer you respect to show you examples of each. Show the code you read to someone else, and see what they think of it.
5. Make a list of your 10 favorite programming tools: the ones you feel you use the most, the ones you almost couldn't live without. Spend an hour reading the docs for one of the tools in your list, chosen at random. In that hour, try learn some new feature of the tool that you weren't aware of, or figure out some new way to use the tool.
6. Pick something you're good at that has nothing to do with programming. Think about how the professionals or great masters of that discipline do their practice. What can you learn from them that you can apply to programming?
7. Get a pile of resumes and a group of reviewers together in a room for an hour. Make sure each resume is looked at by at least three reviewers, who write their initials and a score (1-3). Discuss any resumes that had a wide discrepancy in scoring.
8. Listen in on a [technical phone screen](#). Write up your feedback afterwards, cast your vote, and then talk about the screen with the screener to see if you both reached the same conclusions.
9. Conduct a technical interview with a candidate who's an expert in some field you don't know much about. Ask them to explain it to you from

the ground up, assuming no prior knowledge of that field. Try hard to follow what they're saying, and ask questions as necessary.

10. Get yourself invited to someone else's technical interview. Listen and learn. Try to solve the interview questions in your head while the candidate works on them.
11. Find a buddy for trading practice questions. Ask each other programming questions, alternating weeks. Spend 10 or 15 minutes working on the problem, and 10 or 15 minutes discussing it (finished or not.)
12. When you hear any interview coding question that you haven't solved yourself, go back to your desk and mail the question to yourself as a reminder. Solve it sometime that week, using your favorite programming language.

What I like about Steve's list is that it's somewhat holistic. When some developers think "practice" they can't get beyond code puzzles. But to me, programming is [more about people than code](#), so there's a limit to how much you can grow from solving every obscure programming coding interview problem on the planet.

I also like Peter Norvig's general recommendations for effortful study outlined in [Teach Yourself Programming in Ten Years](#).

1. Talk to other programmers. Read other programs. This is more important than any book or training course.
2. Program! The best kind of learning is learning by doing.
3. Take programming classes at the college or graduate level.
4. Seek out and work on projects with teams of programmers. Find out what it means to be the best programmer on a project—and the worst.
5. Work on projects after other programmers. Learn how to maintain code you didn't write. Learn how to write code so other people can effectively maintain it.
6. Learn different programming languages. Pick languages that have alternate worldviews and programming models unlike what you're used

to.

7. Understand how the hardware affects what you do. Know how long it takes your computer to execute an instruction, fetch a word from memory (with and without a cache miss), transfer data over ethernet (or the internet), read consecutive words from disk, and seek to a new location on disk.

You can also glean some further inspiration from [Pragmatic Dave's 21 Code Katas](#), or maybe you'd like to [join a Coding Dojo](#) in your area.

I don't have a long list of effortful study advice like Steve and Peter and Dave do. I'm far too impatient for that. In fact, there are only two movements in my book of code kata:

1. **Write a blog.** I started this blog in early 2004 as a form of effortful study. From those humble beginnings it has turned into the most significant thing I've ever done in my professional life. So [you should write blogs](#), too. The people who can write and communicate effectively are, all too often, the only people who get heard. They get to set the terms of the debate.
2. **Actively participate in a notable open source project or three.** All the fancy blah blah blah talk is great, but are you [a talker or a doer](#)? This is critically important, because [you will be judged by your actions, not your words](#). Try to leave a trail of public, concrete, useful things in your wake that you can point to and say: I helped build that.

When you can write brilliant code and brilliant prose explaining that code to the world—well, I figure that's **the ultimate code kata**.

[OceanofPDF.com](#)

In Programming, One Is the Loneliest Number

Is software development **an activity preferred by anti-social, misanthropic individuals who'd rather deal with computers than other people?** If so, does it then follow that all software projects are best performed by a single person, working alone?

The answer to the first question may be a reluctant yes, but the answer to the second question is a resounding and definitive no. I was struck by [this beautifully written piece](#) which explains **the dangers of programming alone**:

“Some folks have claimed that [working alone] presents a great opportunity to establish your own process. In my experience, *there is no process in a team of one*. There's nothing in place to hold off the torrents of work that come your way. There's no one to correct you when the urge to gold-plate the code comes along. There's no one to review your code. There's no one to ensure that your code is checked in on time, labeled properly, unit tested regularly. There's no one to ensure that you're following a coding standard. There's no one to monitor your timeliness on defect correction. There's no one to verify that you're not just marking defects as "not reproducible" when, in fact, they are. There's no one to double-check your estimates, and call you on it when you're just yanking something out of your ass.

“There's no one to pick up the slack when you're sick, or away on a business trip. There's no one to help out when you're overworked, sidetracked with phone calls, pointless meetings, and menial tasks that someone springs on you at the last minute and absolutely must be done right now. There's no one to bounce ideas off of, no one to help you figure your way out of a bind, no one to collaborate with on designs, architectures or technologies. You're working in a vacuum. And in a vacuum, no one can hear you scream.

“If anyone's reading this, let this be a lesson to you. Think hard before you accept a job as the sole developer at a company. It's a whole new kind of hell. If given the chance, take the job working with other developers, where you can at least work with others who can mentor you and help you develop your skill set, and keep you abreast of current technology.”

Working alone is a temptation for many desperate software developers who feel trapped, surrounded by incompetence and mismanagement in the desert of the real. Working alone means complete control over a software project, wielding ultimate power over every decision. But working on a software project all by yourself, instead of being empowering, is paradoxically debilitating. It's a shifting mirage that offers the tantalizing promise of relief, while somehow leaving you thirstier and weaker than you started.

Like many programmers, I was drawn to computers as a child because I was an introvert. The world of computers—that calm, rational oasis of ones and zeros—seemed so much more inviting than the irrational, unexplainable world of people and social interactions with no clear right and wrong. Computers weren't better than people, exactly, but they were sure one heck of a lot easier to understand.

Computing in the early, pre-internet era was the very definition of a solitary activity. Dani Berry, the author of M.U.L.E., [sums up with this famous quote](#): "No one ever said on their deathbed, 'Gee, I wish I had spent more time alone with my computer.'" But we've long since left the days of solitary 8-bit programming behind. The internet, and the increasing scope and complexity of software, have made sure of that. I can barely program these days [without an active internet connection](#); I feel crippled when I'm not networked into the vast hive mind of programming knowledge on the internet.

What good are nifty coding tricks if you can't show them off to anyone? How can you possibly learn the craft without being exposed to other programmers with different ideas, different approaches, and different skillsets? Who will review your code and tell you when there's an easier approach you didn't see? **If you're serious about programming, you should demand to work with your peers.**

There's only so far you can go in this field by yourself. Seek out other smart programmers. Work with them. Endeavor to be [the dumbest guy in the room](#), and you'll quickly discover that software development is a far more social activity than most people realize. There's a lot you can learn from your fellow introverts.

OceanofPDF.com

Who's Your Coding Buddy?

I am continually amazed how much better my code becomes after I've had a peer look at it. I don't mean a formal review in a meeting room, or making my code open to anonymous public scrutiny on the internet, or some kind of onerous [pair programming](#) regime. Just **one brief attempt at explaining and showing my code to a fellow programmer**—that's usually all it takes.

This is, of course, nothing new. Karl Wieggers' excellent book “[Peer Reviews in Software: A Practical Guide](#)” has been the definitive guide since 2002.

Addison-Wesley Information Technology Series



Peer Reviews in Software

A Practical Guide

Karl E. Wieggers

I don't think anyone disputes the value of [having another pair of eyes on your code](#), but there's a sort of institutional inertia that prevents it from happening in a lot of shops. In the chapter titled "[A Little Help from Your Friends](#)," Karl explains:

“Busy practitioners are sometimes reluctant to spend time examining a colleague's work. You might be leery of a coworker who asks you to review his code. Does he lack confidence? Does he want you to do his thinking for

him? ‘Anyone who needs his code reviewed shouldn't be getting paid as a software developer,’ scoff some review resisters.

“In a healthy software engineering culture, team members engage their peers to improve the quality of their work and increase their productivity. They understand that the time they spend looking at a colleague's work product is repaid when other team members examine their own deliverables. The best software engineers I have known actively sought out reviewers. Indeed, the input from many reviewers over their careers was part of what made these developers the best.”

In addition to the above chapter, you can sample [Chapter 3](#), courtesy of the author's own [Process Impact](#) website. This isn't just feel-good hand waving. There's actual data behind it. Multiple studies show [code inspections are startlingly effective](#).

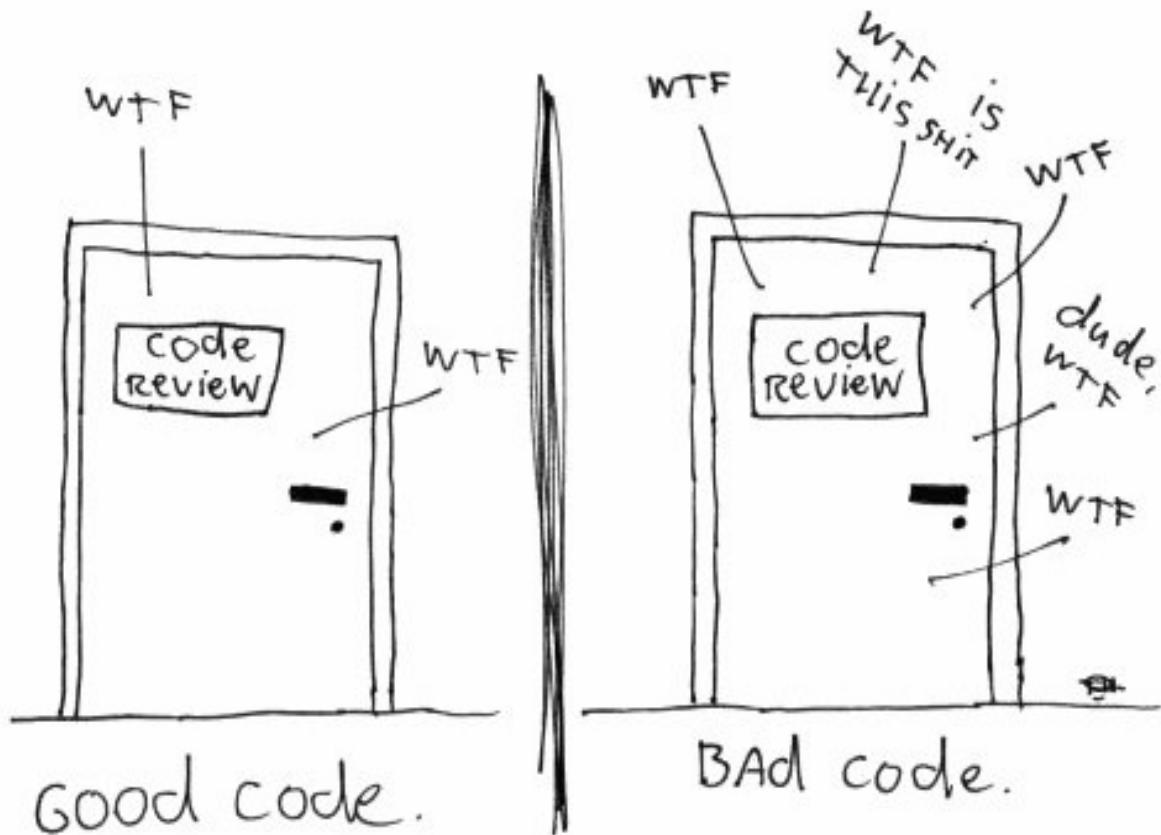
“The average defect detection rate is only 25 percent for unit testing, 35 percent for function testing, and 45 percent for integration testing. In contrast, the average effectiveness of design and code inspections are 55 and 60 percent.”

So why aren't you doing code reviews? Maybe it's because you haven't picked out a coding buddy yet!

Remember those school trips, where everyone was admonished to pick a buddy and stick with them? This was as much to keep everyone out of trouble as safe. Well, the same rule applies when you're building software. Before you check code in, **give it a quick once-over with your buddy**. Can you explain it? Does it make sense? Is there anything you forgot?

I am now required by law to link to [this cartoon](#).

The ONLY VALID MEASUREMENT OF CODE QUALITY: WTFs/MINUTE



(c) 2008 Focus Shift

Thank you, I'll be here all week.

But seriously, this cartoon illustrates exactly the kind of broad reality check we're looking for. It doesn't have to be complicated to be effective. WTFs/minute is a perfectly acceptable unit of measurement to use with your coding buddy. The XP community has promoted [pair programming](#) for years, but I think the buddy system is a far more practical way to achieve the same results.

Besides, who wouldn't want to be **half of an awesome part-time coding dynamic duo?**

That's way more exciting than the prospect of being shackled to the same computer with another person. Think about all the other classic dynamic duos out there:

- Batman and Robin
- [Tango and Cash](#)
- Lennon and McCartney
- Mario and Luigi
- Starsky and Hutch
- Siegfried and Roy
- [Turner and Hooch](#)
- Abbott and Costello
- [Crockett and Tubbs](#)
- Jobs and Wozniak
- Bert and Ernie
- [Ponch and Jon](#)
- Hall and Oates
- Cheech and Chong

Individuals can do great things, but two highly motivated peers can accomplish even more when they work together. Surely there's at least one programmer you work with who you admire or at least respect enough to adopt the buddy system with. (And if not, you might consider [changing your company](#).)

One of the great joys of programming is [not having to do it alone](#). **So who's your coding buddy?**

Software Apprenticeship

In [Software Training Sucks: Why We Need to Roll it Back 1,000 Years](#), Rob Walling makes a **compelling argument for abandoning traditional training classes in favor of apprenticeships**:

[Why not] use the time-tested approach of trades that have been doing it for years? Let's take an electrical apprenticeship as an example: in the United States today, the International Brotherhood of Electrical Workers (I.B.E.W.) trains thousands of electricians every year. They learn through two distinct experiences:

- 1. Attending night school during the week to learn the theory of electricity.*
- 2. Working days on a construction site where they're able to gain experience applying the theory to the hands-on construction of a building*

His first day on the job an apprentice is paired up with a journeyman (an experienced electrician), who shows him the ropes. The journeyman typically talks the apprentice through a task, demonstrates the task, has the apprentice perform the task, then gives feedback. Listen, watch, do, review.



“With software it looks like this: the mentor evaluates the task at hand, be it writing data access code or building a web-based user interface, and holds a white-board discussion with the apprentice (listen). Next, the mentor might write sample code demonstrating a particularly difficult or confusing concept (watch). At this point the mentor sends the mentee off to gain their own experience writing code (do). And finally, the mentor should review the code, providing positive and negative feedback and suggesting improvements (review). Listen, watch, do, review.

“[...] the key to any type of apprenticeship is the ‘do’ step. Most software training gives you the listen and watch, but the ‘do and review’ is what inspires growth and advances skills. The beauty of apprenticeship is that it tackles theory and experience in one fell swoop. And it's easier than you think.”

Instead of a [loose confederation of tribes](#), maybe we should be cultivating apprentice/journeyman/master relationships in software development.

The mixture of theory by night and real world coding by day is particularly compelling. Maybe this is why I've seen so many talented interns turn into amazing developers—they're working on real business code while getting the computer science courseware theory, too.

Being a good mentor isn't easy, though. I have difficulty mentoring developers who are too far apart from me in skill level. I'm too impatient. If you're putting football players together on a field to scrimmage, don't mix professional players with high school players. The skill disparity is too great for them to actually play football together. And how can they learn without playing the game? Now, if you throw some college football players in the mix, it's on!

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

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

OceanofPDF.com

III.

Web Design Principles

OceanofPDF.com

Judging Websites

I was invited to judge the [Rails Rumble](#) last year but was too busy to participate. When they extended the offer again this year, I happily accepted.

“The [Rails Rumble](#) is a distributed programming competition where teams of one to four people, from all over the world, have 48 hours to build an innovative web application, with Ruby on Rails or another Rack-based Ruby web framework. After the 48 hours are up, a panel of expert judges will pick the top ten winners.”

I received an email notifying me that judging begins today, so I cracked my knuckles, sat down in front of my three monitors (all the better to judge with!) and ... saw that there were around **340 entries**.

FoodStrap.me

by weLaika

An easy and fast way to put your food on-line. Load your menu, your customers will book on-line, but they will pay and pick up at your place. A dashboard lets you keep everything under control, and helps you being on-schedule.

LAUNCH SITE FAVORITE LEAVE FEEDBACK

freetime

by dtime + shaine

An application to help people figure out when everyone is free for an event with minimal fuss.

LAUNCH SITE FAVORITE LEAVE FEEDBACK

Socialkart

by AppDreamers

Socialkart lets users do quick comparisons of products from Flipkart, the leading Indian e-commerce site, with their Facebook friends.

LAUNCH SITE FAVORITE LEAVE FEEDBACK

Rango

by Chazinho

Tired of feeling deeply hungry without anything inside refrigerator? Feeling your stomach rumble when neighbors start cooking some delicious food? Enough! With Rango™ you can find a place near you to grab some cheap, delic.

LAUNCH SITE FAVORITE LEAVE FEEDBACK

ScrapsApp

by Prograils

ScrapsApp is code scraps collecting tool (aka snippets) with possibility to share them within a team or designated friend. App is created from need - almost everyday we have problems in office with sharing code - with one.

LAUNCH SITE FAVORITE LEAVE FEEDBACK

That's when I started to get a little freaked out about the math. Perhaps we can throw five percent of the entrants out as obviously incomplete or unfinished. That leaves 323 entries to judge. Personally, I'm not comfortable saying I judged a competition unless I actually look at each one of the

entries, so at an absolute minimum I have to click through to each webapp. Once I do, I couldn't imagine properly evaluating the webapp without spending at least 30 seconds looking at the homepage.

Let's be generous and say I need 10 seconds to orient myself and account for page load times, and 30 seconds to look at each entry. That totals **three and a half hours** of my, y'know, infinitely valuable time. In which I could be finding a cure for cancer, or clicking on LOLcats. I still felt guilty about only allocating half a minute per entry; is it fair to the contestants if I make my decision based on 30 seconds of scanning their landing page and maybe a few desultory clicks?

But then I had an epiphany: **yes, deciding in 30 seconds is totally completely unfair, but that's also exactly how it works in the real world.** Users are going to click through to your web site, look at it for maybe 30 seconds, and either decide that it's worthy, or reach for the almighty back button on their browser and bug out. Thirty seconds might even be a bit generous. In [one Canadian study](#), users made up their mind about websites in under a second.

“Researchers led by Dr. Gitte Lindgaard at Carleton University in Ontario wanted to find out how fast people formed first impressions. They tested users by flashing web pages for 500 milliseconds and 50 milliseconds onto the screen, and had participants rate the pages on various scales. The results at both time intervals were consistent between participants, although the longer display produced more consistent results. Yet, in as little as 50 milliseconds, participants formed judgments about images they glimpsed. The ‘halo effect’ of that emotional first impression carries over to cognitive judgments of a web site's other characteristics including usability and credibility.”

The opportunity cost to switch websites is one tiny little click of the mouse or tap of the finger. What I learned from judging the Rails Rumble most of all is that **your website's front page needs to be kind of awesome.** It is never the complete story, of course, but do not squander your first opportunity to make an impression on a visitor. It may be the only one you get.

I'm not sure I was learning much about these apps while I judged, and for that I am truly sorry. But along the way I accidentally learned a heck of a lot

about **what makes a great front page** for a web application. So I'd like to share that with you, and all future Rails Rumble entrants:

1. **Load reasonably fast.** I've talked about [performance as a feature](#) before; the sooner the front page of your site loads, the sooner I can decide whether or not I am interested. If you are slow, I will resent you for being slow, and the slower you are the more I will resent you for keeping me from not just finding out about you but also keeping me from moving on to the next thing. I need to be an [efficient informant](#). That means moving quickly. Above all else, load fast.
2. **What the %#!@^ is this thing?** The first challenge you have is not coding your app. It is explaining what problem your app solves, and why anyone in the world would possibly care about that. You need [an elevator pitch](#) on your front page: can you explain to a complete stranger, in 30 seconds, why your application exists? Yes, writing succinctly and clearly is an art, but keep pounding on that copy, keep explaining it over and over and over until you have your explanation polished to the fine sheen of a diamond. When you're confident you could walk up to any random person on the street, strike up a conversation about what you're working on, and not have their eyes gloss over in boredom and/or fear—that's when you're ready. That's the text you want on your home page.
3. **Show me an example.** Okay, so you're building the ultimate tool for cataloging and sharing Beanie Babies on Facebook. Awesome, let me be an angel investor in your project so I can get me a piece of those sweet, sweet future billions. The idea is sound. But everyone knows that [ideas are worthless, whereas execution is everything](#). I have no clue what the execution of your idea is unless you show it to me. At the very least throw up some screenshots of what it would look like if I used your webapp, with some juicy real world examples. And please, please, please, for the love of God please, do not make me sign up, click through a video, watch a slideshow, or any of that nonsense. Only emperors and princes have that kind of time, man. [Show, don't tell](#).
4. **Give me a clear, barrier-free call to action.** In the rare cases where the app passes the above three tests with flying colors, I'm invested: I am now willing to spend even more of my time checking it out. What

do I do next? Where do I go? Your job is to make this easy for me. I call this "the put a big-ass giant obvious fluorescent lime green button on your home page" rule. You can have more than one, but I'd draw the line at two. And make the text on the button descriptive, like Start sharing your favorite Beanie Babies → or Build your dream furry costume →. If you require login at this point, I strongly urge you to skip that barrier and have a live sample I can view without logging in at all, just to get a taste of how things might work. If you're really, really slick you will make it seamless to go from an unregistered to a registered state without losing anything I've done.

5. **Embrace your audience, even if it means excluding other audiences.** Even if you nail all the above, you might not fit into my interest zone through absolutely no fault of your own. If you built the world's most innovative and utterly disruptive Web 5.0 Pokédex, there's a lot of people who won't care one iota about it, because they're [not really into Pokemon](#). This is not your fault and it is certainly not their fault. You need to embrace the idea that half of all success is knowing your core audience and not trying to water it down so much that it appeals to "everyone." Don't patronize me by trying to sell me on the idea that everyone should care about babies, or invoicing, or sports, or being a student, or whatever. Only the people who need to care will care, and that's who you are talking to. So have the confidence to act like it.

I realize that Rails Rumble apps only have a mere 48 hours to build an entire app from scratch. I am not expecting a super professional amazing home page on every one of the entries, nor did I judge it that way. But I do know that **a basic sketch of a homepage design is the first thing you should work on in any webapp, because it serves as the essential starting design document and vision statement.** Unless you start with a basic homepage that meets the above five rules, your app won't survive most judges, much less the [herds of informavores](#) running wild on the Internet.

OceanofPDF.com

In Pursuit of Simplicity

John Maeda created quite a stir with his montage of the Yahoo and Google homepages from 1996 to 2006 in [simple is about staying simple](#):



Although Philipp Lenssen has posted on this topic before (he calls it [the portal plague](#)), it's still striking. Altavista made [the same mistake](#), and they didn't survive.

There's an interesting anecdote about Google's absolute focus on minimalism in Seth Godin's book [Purple Cow](#):

“It turns out that the folks at Google are obsessed with the email they get criticizing the service. They take it very seriously. One person writes in every once and a while and he never signs his name. According to Marissa Meyer at Google, ‘Every time he writes, the e-mail contains only a two-digit number. It took us a while to figure out what he was doing. He's counting the number of words on the home page. When the number goes up, he gets irritated, and e-mails us the new word count. As crazy as it sounds, his emails are helpful, because they put an interesting discipline on the UI team not to introduce too many links. It's like a scale that tells you that you've gained two pounds.’”

And of course, [37signals](#) is famous for their mantra of [less as a competitive advantage](#):

“Conventional wisdom says to beat your competitors you need to one-up them. If they have 4 features, you need 5. Or 15. Or 25. If they're spending X, you need to spend XX. If they have 20, you need 30.

“While this strategy may still work for some, it's expensive, resource intensive, difficult, defensive, and not very satisfying. And I don't think it's good for customers either. It's a very Cold War mentality—always trying to one-up. When everyone tries to one-up, we all end up with too much. There's already too much ‘more’—what we need are simple solutions to simple, common problems, not huger solutions to huger problems.

“What I'd like to suggest is a different approach. Instead of one-upping, try one-downing. Instead of outdoing, try underdoing. Do less than your competitors to beat them.”

Usability guru Donald Norman thinks the comparison between Google and Yahoo is misleading, and offers [the truth about Google's so-called "simplicity"](#):

“Is Google simple? No. Google is deceptive. It hides all the complexity by simply showing one search box on the main page. The main difference, is that if you want to do anything else, the other search engines let you do it from their home pages, whereas Google makes you search through other, much more complex pages. Why aren't many of these just linked together? Why isn't Google a unified application? Why are there so many odd, apparently free-standing services?”

I think this is a completely wrongheaded analysis, because **I don't want to do anything else**. All I want is to find what I'm searching for. Like Damien Katz, I believe [features don't matter](#):

“These people don't care about your flexible, brilliant architecture. They don't wish to tweak settings. They don't want to spend more than 10 consecutive seconds confused. They just want simple, they want to get their task done and move on. They don't want to spend time learning anything because they know they'll probably just forget it long before they'll need to do it again anyway.”

We should always be **in pursuit of simplicity**, in whatever form it takes.

OceanofPDF.com

Will Apps Kill Websites?

I've been an eBay user since 1999, and I still frequent eBay as both buyer and seller. In that time, eBay has transformed from a place where geeks [sell broken laser pointers to each other](#), into a global marketplace where businesses sell anything and everything to customers. If you're looking for strange or obscure items, things almost nobody sells new any more, or grey market items for cheap, eBay is still not a bad place to look.

At least for me, eBay still basically works, after all these years. But one thing hasn't changed: **the eBay website has always been difficult to use and navigate**. They've updated the website recently to remove some of the more egregious cruft, but it's still way too complicated. I guess I had kind of accepted old, complex websites as the status quo, because I didn't realize how bad it had gotten until I compared the experience on the eBay website with the experience of the eBay apps for mobile and tablet.

eBay Website



pee wee herman doll All Categories Search Advanced Include description

70 results found for pee wee herman doll Save search Tell Us What You Think?

Categories

- Toys & Hobbies (61)
- TV, Movie & Character Toys (52)
- Vintage & Antique Toys (4)
- Action Figures (3)
- Models & Kits (1)
- Classic Toys (1)
- Dolls & Bears (6)
- Dolls (6)
- See all categories

Condition

- New (21)
- Used (42)
- Not Specified (7)
- Choose more...

Price

\$ to \$

Seller

- eBay Top-rated sellers
- Specify sellers...

Buying formats

- Auction
- Buy It Now
- Choose more...

Show only

- Completed listings
- Choose more...

Location

- US Only
- North America
- Worldwide


All items Auctions only Buy It Now Products & reviews **Beta** Customize view

View as: Sort by: Best Match Page 1 of 2







| | | | | |
|--|--|------------------|--------------------|------------|
| | Talking Pee-Wee Herman Doll Returns: Not accepted | 3 bids | \$7.50 | 4h 49m |
| | 1967 sealed, NRFB mint 18" Pee Wee Herman Talking Doll, works great, great box, 1st Returns: Accepted within 14 days | Top-rated seller | Buy It Now \$64.75 | 27d 2h 36m |
| | PEE-WEE HERMAN DOLL PREOWNED HAS WEAR 17 3/4 INCH 1967 MATCHBOX TOYS READ! Returns: Accepted within 7 days | Top-rated seller | Buy It Now \$9.99 | 1d 20h 48m |
| | 1967 Pee-Wee Herman Pull String Talking Doll - 17" Tall Returns: Accepted within 7 days | 2 bids | \$20.49 | 8h |
| | VINTAGE PEE WEE HERMAN PULL STRING DOLL 1967 MATCHBOX C426 One-day shipping available Returns: Accepted within 14 days | Top-rated seller | Buy It Now \$24.99 | 15d 21h 6m |
| | Pee Wee Herman 16 inch talking doll | 0 bids | \$19.00 Free | 8h 51m |

eBay Mobile App

Search  Refine

All Auction Buy It Now

70 items found for "pee wee herman..."

| | | | |
|---|--|---|---|
|  | Talking Pee-Wee Herman Doll | \$7.50 + \$5.99 3 bids 4h 32m | > |
|  | 1987 sealed, NRFB mint 18" Pee Wee Herman Talking Doll, works great, great | \$64.75 + \$17.50 Buy It Now 27d 2h | > |
|  | 1987 Pee-Wee Herman Pull String Talking Doll - 17" Tall | \$20.49 + \$10.40 2 bids 7h 43m | > |
|  | Pee Wee Herman 18 inch talking doll | \$19.00 Free | > |










Home Search My eBay Sell Settings

eBay Tablet App

70 Search Results

pee wee herman ...

All Categories All Listings Sort by Best Match Refine

| | | |
|--|--|--|
|  <p>Talking Pee-Wee Herman Doll</p> <p>3 bids 4h 24m</p> <p>\$7.50 + \$5.99</p> |  <p>1987 sealed, NRFB mint 18" Pee Wee Herman Talking Doll, works great, great...</p> <p>Buy It Now 27d 2h</p> <p>\$64.75 + \$17.50</p> |  <p>1987 Pee-Wee Herman Pull String Talking Doll - 17" Tall</p> <p>2 bids 7h 35m</p> <p>\$20.49 + \$10.40</p> |
|  <p>Pee Wee Herman 18 inch talking doll</p> <p>0 bids 8h 26m</p> <p>\$19.00 FREE SHIPPING</p> |  <p>pee-wee herman pull string doll 1989 in unopened box still talks</p> <p>0 bids 1d 1h</p> <p>\$75.00 + \$17.85</p> |  <p>PEE-WEE HERMAN DOLL PREOWNED HAS WEAR 17 3/4 INCH 1987 MATCHB...</p> <p>Buy It Now 1d 20h</p> <p>\$9.99 + \$8.83</p> |
|  <p>VINTAGE PEE WEE HERMAN TALKING DOLL 1987 MATCHBOX</p> <p>0 bids 1d 1h</p> <p>\$40.00</p> |  <p>RARE 1987 14" Matchbox Pee Wee Herman Chairy Hand Puppet w/ 18" Pe...</p> <p>0 bids 1d 1h</p> <p>\$40.00</p> |  <p>Pee Wee Herman Doll 17 inch</p> <p>0 bids 1d 1h</p> <p>\$9.99</p> |

Unless you're some kind of super advanced eBay user, you should probably avoid the website. The tablet and mobile eBay apps are just plain simpler,

easier, and faster to use than the eBay website itself. I know this intuitively from using eBay on my devices and computers, but there's also [usability studies](#) with data to prove it, too. To be fair, eBay is struggling under the massive accumulated design debt of a website originally conceived in the late 90s, whereas their mobile and tablet app experiences are recent inventions. **It's not so much that the eBay apps are great, but that the eBay website is so very,very bad.**

The implied lesson here is to **embrace constraints**. Having a limited, fixed palette of UI controls and screen space is a strength. A strength we used to have in early Mac and Windows apps, but seem to have lost somewhere along the way as applications got more powerful and complicated. And it's endemic on the web as well, where the eBay website has been slowly accreting more and more functionality since 1999. The nearly unlimited freedom that you get in a modern web browser to build whatever UI you can dream up, and assume as large or as small a page as you like, often ends up being harmful to users. It certainly is in the case of eBay.

If you're starting from scratch, you should always [design the UI first](#), but now that we have such great mobile and tablet device options, consider designing your site for the devices that have the strictest constraints first, too. This is the thinking that led to [mobile first design strategy](#). It helps you stay focused on a simple and uncluttered UI that you can scale up to bigger and beefier devices. Maybe eBay is just going in the wrong direction here; **design simple things that scale up; not complicated things you need to scale down.**

[Above all else, simplify!](#) But why stop there? If building the mobile and tablet apps first for a web property produces a better user experience—why do we need the website, again? Do great tablet and phone applications make websites obsolete?

Why are apps better than websites?

1. **They can be faster.** No browser overhead of CSS and HTML and JavaScript hacks, just pure native UI elements retrieving precisely the data they need to display what the user requests.

2. **They use simple, native UI controls.** Rather than imagineering whatever UI designers and programmers can dream up, why not pick from a well understood palette of built-in UI controls on that tablet or phone, all built for optimal utility and affordance on that particular device?
3. **They make better use of screen space.** Because designers have to fit just the important things on four inch diagonal mobile screens, or 10 inch diagonal tablet screens, they're less likely to fill the display up with a bunch of irrelevant noise or design flourishes (or, uh, advertisements). Just the important stuff, thanks!
4. **They work better on the go and even offline.** In a mobile world, you can't assume that the user has a super fast, totally reliable Internet connection. So you learn to design apps that download just the data they need at the time they need to display it, and have sane strategies for loading partial content and images as they arrive. That's assuming they arrive at all. You probably also build in some sort of offline mode, too, when you're on the go and you don't have connectivity.

Why are websites better than apps?

1. **They work on any device with a browser.** Websites are as close to universal as we may ever get in the world of software. Provided you have a HTML5 compliant browser, you can run an entire universe of "apps" on your device from day zero, just by visiting a link, exactly the same way everyone has on the Internet since 1995. You don't have to hope and pray a development community emerges and is willing to build whatever app your users need.
2. **They don't have to be installed.** Applications, unlike websites, can't be visited. They aren't indexed by Google. Nor do applications magically appear on your device; they must be explicitly installed. Even if installation is a one-click affair, your users will have to discover the app before they can even begin to install it. And once installed, they'll have to [manage all those applications like so many Pokemon](#).
3. **They don't have to be updated.** Websites are always on the [infinite version](#). But once you have an application installed on your device, how

do you update it to add features or fix bugs? How do users even know if your app is out of date or needs updating? And why should they need to care in the first place?

4. **They offer a common experience.** If your app and the website behave radically differently, you're forcing users to learn two different interfaces. How many different devices and apps do you plan to build, and how consistent will they be? You now have a community divided among many different experiences, fragmenting your user base. But with a website that has a decent mobile experience baked in, you can deliver a consistent, and hopefully consistently great, experience across all devices to all your users.

I don't think there's a clear winner, only pros and cons. But **apps will always need websites**, if for nothing else other than a source of data, as a mothership to phone home to, and a place to host the application downloads for various devices.

And if you're obliged to build a website, why not build it out so it offers a reasonable experience on a mobile or tablet web browser, too? I have nothing against a premium experience optimized to a particular device, but shouldn't all your users have a premium experience? eBay's problem here isn't mobile or tablets per se, but that they've let their core web experience atrophy so badly. I understand that there's a lot of inertia around legacy eBay tools and long time users, so it's easy for me to propose radical changes to the website here on the outside. Maybe the only way eBay can redesign at all is on new platforms.

Will mobile and tablet apps kill websites? A few, certainly. **But only the websites stupid enough to let them.**

OceanofPDF.com

Doing It Like Everybody Else

[Jon Galloway](#) called me out in a comment yesterday for [advocating a non-standard approach](#):

“Web forms have become a convention, and users have been trained for 10 years on how to fill out forms. Users would get confused, and some would bail out (abandon carts, etc.). Web forms work, and we know how to use them. Your form example violates the ‘[Don't Make Me Think](#)’ principle on many levels.”

In a sense, he's right. When it comes to coding, as [Steve Rowe points out](#), **always favor consistency over cleverness**:

“The class isn't the main point of this post, however. Rather, it is some advice that Peter gave a few times during the class. Someone might ask a question like ‘Can't I do x in some funky way?’ and he would answer, ‘You could, but no one would expect to see it so don't.’ The point he was making is that we, as programmers, should stay away from being clever. We should, as much as possible, try to do things the same way everyone else does them. Why? Because you won't be the only person to work on this code. Even if you are, the next time you touch it might be a year or two from now. If you did something clever, the next person to touch it will look at the code and not immediately understand. This will have one of two consequences. Either they will have to spend 10 minutes just trying to understand what it is you did or, worse, they will assume you made a mistake and ‘fix’ it by making it less clever. Neither of these results is desirable. Unless you are writing one-off code for yourself you need to write it in a manner to make it easily understandable so that it can be easily maintained.”

It's clearly a bad idea to write code with a "how 'bout we try it this way" mentality, as [humorously noted by Alex Papadimoulis](#):

"A client has asked me to build and install a custom shelving system. I'm at the point where I need to nail it, but I'm not sure what to use to pound the nails in. Should I use an old shoe or a glass bottle?"

a) It depends. If you are looking to pound a small (20 pound) nail in something like drywall, you'll find it much easier to use the bottle, especially if the shoe is dirty. However, if you are trying to drive a heavy nail into some wood, go with the shoe: the bottle will shatter in your hand.

b) There is something fundamentally wrong with the way you are building; you need to use real tools. Yes, it may involve a trip to the toolbox (or even to the hardware store), but doing it the right way is going to save a lot of time, money, and aggravation through the lifecycle of your product. You need to stop building things for money until you understand the basics of construction.”

However, when it comes to issues of user interface, consistency isn't always a virtue. **User interfaces should be internally consistent, but not necessarily consistent with every other application in the rest of the world.** That said, some UI elements become so ingrained into popular culture that they should be followed for consistency's sake. Some good examples are:

- A search box in the upper-right hand corner
- A logo in the upper-left hand corner that takes you back home
- The "forward" and "back" buttons

But not all user interface conventions are created equal. Some are timeless. Some are there by default, because nobody bothered to sufficiently question them. Some grow old and outlive their usefulness. **How do we discriminate between conventions that actually help us and those that are merely... expected?**

The answer, of course, is to try multiple approaches and collect usage data to determine what works and what doesn't. This is (relatively) easy for web apps, which is why [Amazon](#), [Yahoo](#) and [Google](#) are all notorious for doing it. They'll serve up experimental features to a tiny fraction of the user base, collect data on how those features are used, then feed that back into their decision making process.

If we built UI with an iron-clad guarantee that we would "do it like everyone else," would we have ever experienced the ultra simple Mom-

friendly [Tivo UI](#)? Or Windows Media Center's amazing, utterly un-Windows-like ten foot UI? Would Office 12 be using the [innovative new ribbon](#) instead of traditional toolbars and menus? Heck, would we have ever made the transition from character mode to GUIs?

I think **UI experimentation is not only desirable, but necessary**. If we don't experiment, we can't evolve UI forward. However, you have to do it the right way:

1. Have a complete understanding of the current convention and how it arose
2. Have a good, reasoned argument for deviating from the convention
3. Collect usage data on your experiments
4. Make decisions based on the usage data

If you're not collecting usage data, or your reason is "it looks better this way," then you're doing it wrong, and you should stick with the conventions.

OceanofPDF.com

The One-Button Mystique

I [enjoy my iPhone](#), but I can't quite come to terms with one aspect of its design: Apple's insistence that there can be only ever be one, and only one, button on the front of the device.

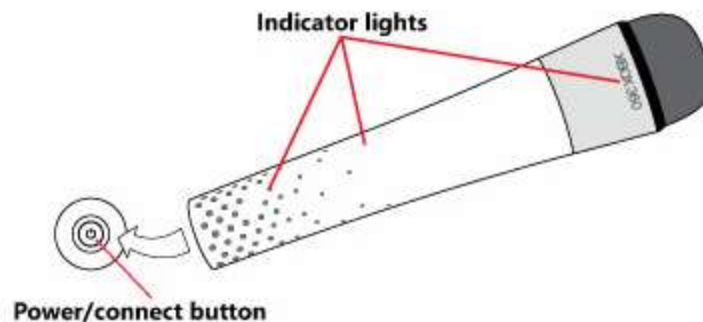


I also own a [completely button-less Kindle Fire](#), and you'll get no argument from me that **there should be at least one obvious "Jesus Handle" button on the front of any gadget**. I do wonder why Amazon decided to make the Fire buttonless, when every other Kindle they ship has a home button. Amazon has a track record of making some awfully rough version 1.0 devices; I'm sure they'll add a home button in a version or two. And, hey, at [only \\$199](#), I'm willing to cut them a little slack. For now.

Even Apple is no stranger to buttonless devices. Consider the [oddly buttonless third generation iPod Shuffle](#), where you had to double and even triple click the controls on the headphones to do basic things like advance tracks. Oh, and by the way, this also made every set of headphones you own obsolete, at least for use with this model. The fourth gen shuffle rapidly switched back to physical controls on the device, and the fifth gen went to touch controls on the device, as expected.



Microsoft is just as guilty. I sometimes struggle with the otherwise awesome [Xbox 360 Wireless Microphone](#). It has only a power button and some lights.



In its defense, for the most part it does just work when you pick it up and start singing (badly, in my case), but I admit to being slightly perplexed every time I have to sync it with an Xbox, or figure out what's going on with it. [Can you blame me?](#)

When you turn on the microphone, the built-in lights shine to display the microphone status as follows:

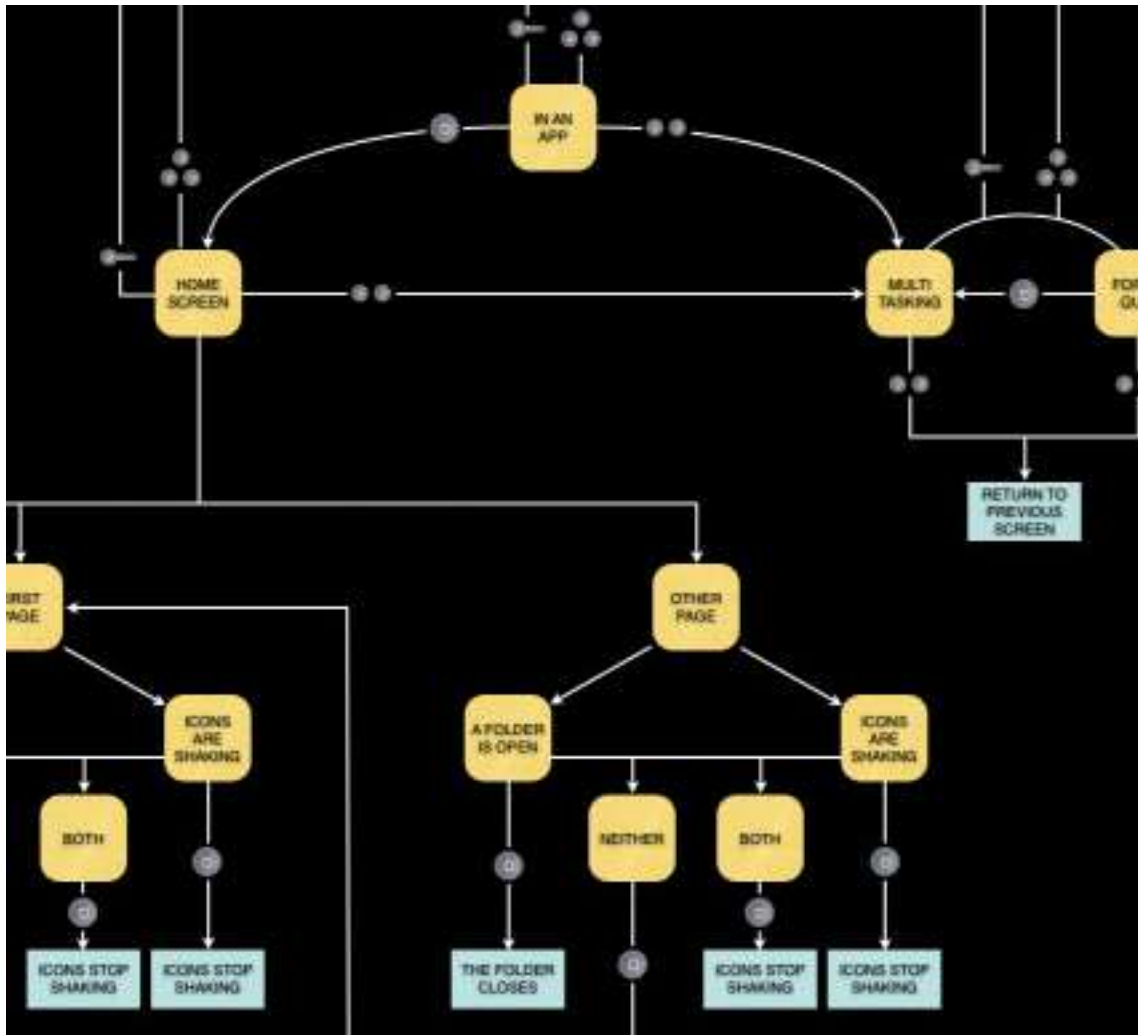
- *Power on: lights flash green one time every second*
- *Connecting: lights flash green four times every second*
- *Connection complete: lights flash blue, and then stops*

When your battery power is low, the built-in lights shine to display the battery charge status as follows:

- *Low: Lights flash amber one time every three seconds*
- *Critical: Lights flash amber one time every second*

When your microphone moves out of the wireless range of your console, the lights flash green one time every second. The lights can also change color together with supported game titles.

If we can agree that no buttons is clearly a bad idea, I think it follows that one button is problematic in its own way. I have the same issue with the single button on the iPhone that [I do with the single button mouse](#)—it may be OK-ish at the very beginning, but over time it leads to absurd, almost comical overloading of functionality. **Consider how many different things the single button on the face of an iPhone now controls:**



(diagram courtesy [Andrew Durdin](#), [source](#))

The iPhone home button? Why, it's easy! You have your choice of...

- single-click
- double-click
- triple-click
- click and hold
- click and pause and click again

All of which have different meanings at different times, of course. In particular I spend a lot of time double-clicking to get to the active apps list,

and I often mis-tap which kicks me over to the home screen. I have so many apps installed on my iPhone that search is the only rational way to navigate. This means I search a lot, which requires clicking once to get to the default home page, pausing, then clicking again. Sometimes I click too long, which is then detected as click-and-hold, and I get the voice search app which I am... er, [not a fan of](#), to put it mildly.

I've gotten to the point where I dread using the home button on my iPhone because it [Makes Me Think](#). And I get it wrong a significant percentage of the time. This isn't the way it's supposed to be.

You might be expecting me to turn into a rabid Windows Phone or Android fanboy about now and snarkily note how they get it right. I'm not sure they do. Either of them. [They all manage to suck in their own special way.](#)



When there's one button on the device, at least it's clear what that button is supposed to do, right? Well, [sometimes](#).

There is one theme I agree with here—**the clearly marked back button on both Android and Windows phones, just like a web browser**. I mostly use my iPhone as a platform for the Internet, and the simplicity of the Internet is its primary strength: a collection of obvious things to click, and an easy, giant honking back button so you never get lost in its maze of twisty passages, all alike. It is true that browsers have a home button, but the latest versions of Chrome, Firefox, and Internet Explorer have all but pushed that home button off the UI in favor of the ginormous back button. While I'll tentatively agree that not all phone apps have to behave like the Internet, the Internet is becoming more and more of [a platform for bona fide applications](#) every day. The back button is a UI paradigm that works like gangbusters for webapps, and I'd argue strongly in favor of that being a hard button on a device.

But once you add three buttons, thinking starts to creep in again. Am I pressing the correct button? That's never good. And I don't even know what that third button is supposed to be on the Android phone! I could possibly be in favor of the hard search button on the Windows phone, I suppose, but I'd rather see good, consistent use of two buttons on the face of a device before [willy-nilly adding Yet Another Button](#). I think there's a reason the industry has more or less standardized on a two-button mouse, for example. (Yes, there is [that pesky middle button](#), but it's a nice to have, not an essential part of the experience.)

What about the one finger solution? Even with touch devices, **one finger does not seem to be enough**; there's a [curious overloading of the experience](#) over time.

“On the iPad, there are a number of system-wide gestures, such as swiping left or right with four fingers to switch between apps. Four-finger swipes? That's convoluted, but imagine a virtual mixing console with horizontal sliders. Quickly move four of them at once...and you switch apps. Application designers have to work around these, making sure that legitimate input methods don't mimic the system-level gestures.

“The worst offender is this: swipe down from the top of the screen to reveal the Notification Center (a window containing calendar appointments, the weather, etc.). A single-finger vertical motion is hardly unusual, and many apps expect such input. The games Flight Control and Fruit Ninja are two prime examples. Unintentionally pulling down the Notification Center during normal gameplay is common. A centered vertical swipe is natural in any paint program, too. Do app designers need build around allowing such controls? Apparently, yes.”

Yes, our old friend overloading is now on the touch scene in spades: for all but the simplest use of a tablet, you will inevitably find yourself double-tapping, tapping and holding, swiping with two fingers, and so on.

Apple's done a great job of embodying simplicity and clean design, but I often think they go too far, particularly at the beginning. For example, the first Mac [didn't even have cursor keys](#). Everything's a design call, and somewhat subjective, but like Goldilocks, I'm going to maintain that the secret sauce here is not to get the porridge too cold (no buttons) or too hot

(3 or more buttons), but just right. I'd certainly be a happier iPhone user if I didn't have to think so much about what's going to happen when I press my home button for the hundredth time in a day.

OceanofPDF.com

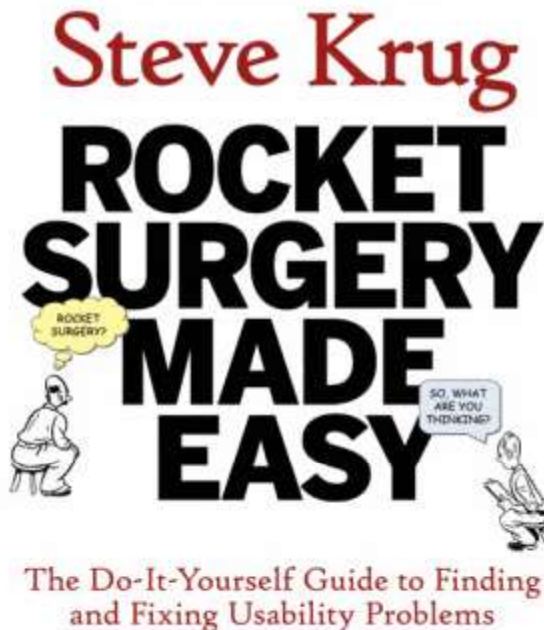
Usability on the Cheap and Easy

Writing code? That's the easy part. Getting your application [in the hands of users](#), and creating applications that [people actually want to use](#)—now that's the hard stuff.

I've been a long time fan of Krug's book [Don't Make Me Think](#). Not just because it's a quick, easy read (and it is!)—but because it's the most concise and most approachable book I've ever found to teach the fundamental importance of usability. As far as I'm concerned, if you want to help us make the software industry a saner place, the first step is getting [Don't Make Me Think](#) in the hands of as many of your coworkers as you can. **If you don't have people that care about usability on your project, your project is doomed.**

Beyond getting people over the hurdle of at least paging through the Krug book, and perhaps begrudgingly conceding that this usability stuff matters, the next challenge is figuring out how to integrate usability testing into your project. It's easy to say "Usability is Important!" but you have to walk the walk, too. I touched on some low friction ways to get started in [Low-Fi Usability Testing](#). That rough outline is now available in handy, more complete book form—“[Rocket Surgery Made Easy: The Do-It-Yourself Guide to Finding and Fixing Usability Problems](#).”

The how-to companion to the bestselling *Don't Make Me Think!*
A Common Sense Approach to Web Usability



Don't worry, Krug's book is just as usable as his advice. It's yet another quick, easy read. Take it from the man himself:

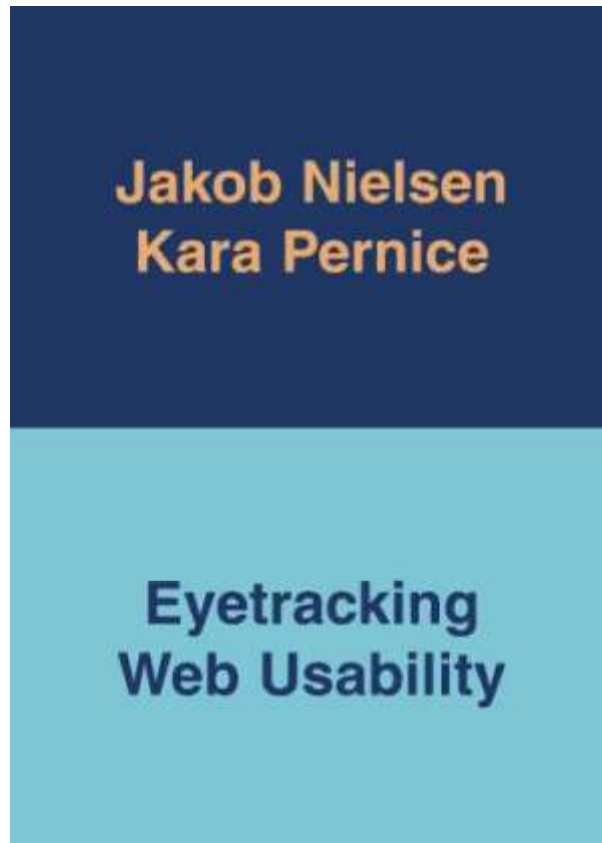
- **Usability testing is one of the best things people can do to improve Web sites (or almost anything they are creating that people have to interact with).**
- Since most organizations can afford to hire someone to do testing for them on a regular basis, everyone should learn to do it themselves. And ...
- I could probably write a pretty good book explaining how to do it.

If you're wondering what the beginner's "how do I boil water?" recipe for software project usability is, stop reading this post and get a copy of [Rocket Surgery Made Easy](#). Now.

One of the holy grails of usability testing is [eyetracking](#)—measuring where people's eyes look as they use software and web pages. Yes, there are clever JavaScript tools that can measure where users move their pointers, but that's only a small part of the story. Where the eye wanders, the pointer may not,

and vice-versa. But, who has the time and equipment necessary to conduct an actual eyetracking study? Almost nobody.

That's where [Eyetracking Web Usability](#) comes in.



Eyetracking Web Usability is chock full of incredibly detailed eyetracking data for dozens of websites. Even though you (probably) can't afford to do real eyetracking, you can certainly use this book as a reference. There is enough variety in UI and data that you can map the results, observations, and explanations found here to what your project is doing.

This particular book is rather eyetracking specific, but it's just the latest entry in [a whole series on usability](#), and I recommend them all highly. These books are a fount of worthwhile data for anyone who works on software and cares about usability, from one of the most preeminent usability experts on the web.

Usability isn't really cheap or easy. It's an endless war, with innumerable battlegrounds, stretching all the way back to the dawn of computing. But these books, at least, are cheap and easy in the sense that they give you

some **basic training in fighting the good (usability) fight**. That's the best I can do, and it's all I'd ask from anyone else I work with.

OceanofPDF.com

The Opposite of Fitts' Law

If you've ever wrangled a user interface, you've probably heard of [Fitts' Law](#). It's pretty simple—**the larger an item is, and the closer it is to your cursor, the easier it is to click on**. Kevin Hale put together [a great visual summary of Fitts' Law](#), so rather than over-explain it, I'll refer you there.

The short version of Fitts' law, to save you all that tedious reading, is this:

- Put commonly accessed UI elements on the edges of the screen. Because the cursor automatically stops at the edges, they will be easier to click on.
- Make clickable areas as large as you can. Larger targets are easier to click on.

I know, it's very simple, almost too simple, but humor me by following along with some thought exercises. Imagine yourself trying to click on ...

- a 1 x 1 target at a random location
- a 5 x 5 target at a random location
- a 50 x 50 target at a random location
- a 5 x 5 target in the corner of your screen
- a 1 x 100 target at the bottom of your screen

Fitts' Law is mostly common sense, and enjoys enough currency with UI designers that they're likely to know about it even if [they don't follow it as religiously as they should](#). Unfortunately, I've found that designers are much less likely to consider the opposite of Fitts' Law, which is arguably just as important.

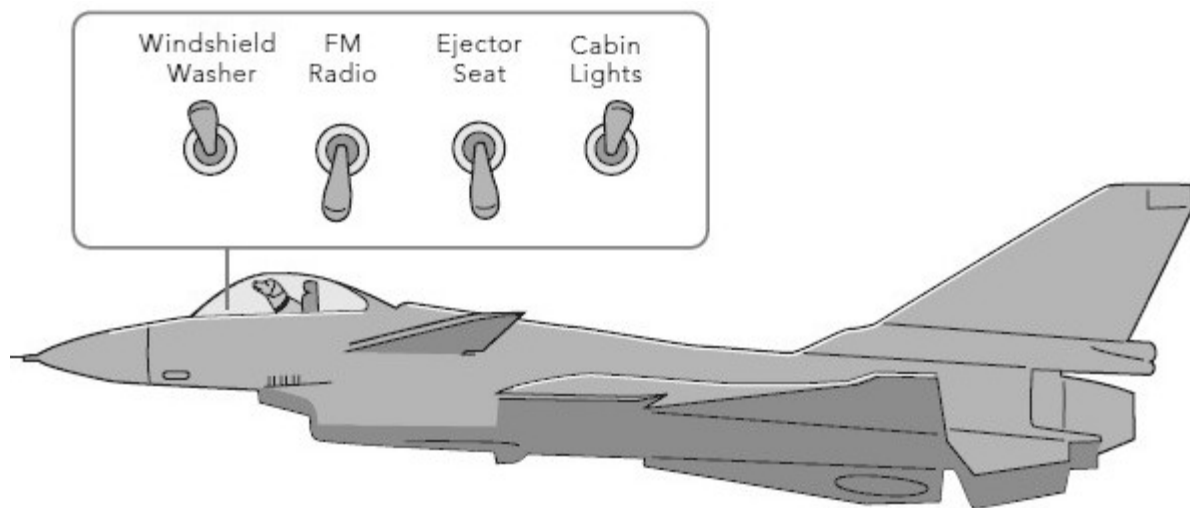
If we should make UI elements we want users to click on large, and ideally place them at corners or edges for maximum clickability—**what should we**

do with UI elements we don't want users to click on? Like, say, the "delete all my work" button?

Alan Cooper, in [About Face 3](#), calls this the ejector seat lever.

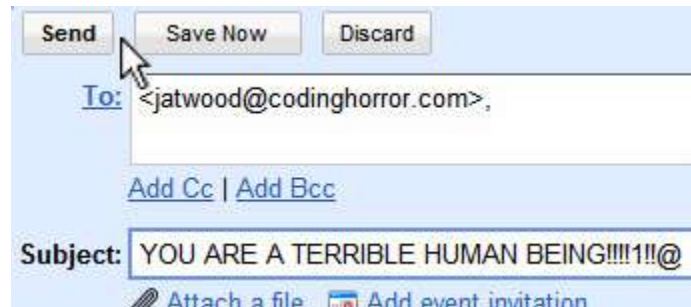
“In the cockpit of every jet fighter is a brightly painted lever that, when pulled, fires a small rocket engine underneath the pilot's seat, blowing the pilot, still in his seat, out of the aircraft to parachute safely to earth. Ejector seat levers can only be used once, and their consequences are significant and irreversible.

“Applications must have ejector seat levers so that users can occasionally move persistent objects in the interface, or dramatically (sometimes irreversibly) alter the function or behavior of the application. The one thing that must never happen is accidental deployment of the ejector seat.”



“The interface design must assure that a user can never inadvertently fire the ejector seat when all he wants to do is make some minor adjustment to the program.”

I can think of a half-dozen applications I regularly use where **the ejector seat button is inexplicably placed right next to the cabin lights button**. Let's take a look at our old friend GMail, for example:



I can tell what you're thinking. Did he click **Send** or **Save Now**? Well, to tell you the truth, in all the excitement of composing that angry email, I kind of lost track myself. Good thing we can easily undo a sent mail! Oh wait, we totally can't. Consider my seat, or at least that particular rash email, ejected.

It's even worse when I'm archiving emails.



While there were at least 10 pixels between the buttons in the previous example, here there are all of... three. Every few days I accidentally click **Report Spam** when I really meant to click **Archive**. Now, to Google's credit, they do offer a simple, obvious undo path for these accidental clicks. But I can't help wondering why it is, exactly, that these two buttons with such radically different functionality just have to be right next to each other.

Undo is powerful stuff, but wouldn't it be better still if I wasn't pulling the darn ejector seat lever all the time? Wouldn't it make more sense to put that risky ejector seat lever in a different location, and make it smaller? Consider the WordPress post editor.



Here, the common **Update** operation is large and obviously a button—it's easy to see and easy to click on. The less common **Move to Trash** operation is smaller, presented as a vanilla hyperlink, and placed well away from Update.

The next time you're constructing a user interface, you should absolutely follow Fitts' law. It just makes sense. But don't forget to follow the opposite of Fitts' law, too—uncommon or dangerous UI items should be difficult to click on!

OceanofPDF.com

Usability vs. Learnability

In this [1996 Alertbox](#), Jakob Nielsen champions writing for the web in an inverted pyramid style:

“Journalists have long adhered to the inverse approach: start the article by telling the reader the conclusion (‘After long debate, the Assembly voted to increase state taxes by 10 percent’), follow by the most important supporting information, and end by giving the background. This style is known as the inverted pyramid for the simple reason that it turns the traditional pyramid style around. Inverted-pyramid writing is useful for newspapers because readers can stop at any time and will still get the most important parts of the article.

“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. Very interested readers will scroll, and these few motivated souls will reach the foundation of the pyramid and get the full story in all its gory detail.”

For some reason, the idea that "content should never appear below the fold because users don't scroll" was a very widely held belief well into the millennium. I don't know what user studies Mr. Nielsen is referring to; even the ancient 1998 reference “[Web Site Usability: A Designer's Guide](#)” found no evidence to support this claim:

“The Fidelity site, more than any other site we tested, went to great lengths to have many of its pages completely above the fold. The result is lots of pages, each with small amounts of content. There was no evidence to suggest that this strategy helped or hurt.

“In fact, we never saw any user frustration with scrolling. For instance, when we counted ‘first clicks’—the first place people clicked when they came to a new site—clicks were just as likely to be above the fold as they were to be below it. If scrolling below the fold was a source of frustration, we would have expected to see some sort of negative correlation between first clicks below the fold and success, but we didn't.”

In 2003, Jakob added an addendum to his Alertbox, which reads:

“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.”

You should certainly try to put the most important information at the top of whatever it is you're writing, be it a website, a program, an email, a resume, etc. Believe me, I've learned this the hard way; you're lucky if they [read anything](#), much less the first paragraph.

But to claim that users don't scroll is downright ridiculous, even for 1996. Let's say you had a user who didn't know how to scroll a web page. How long would it take this user, however timid they may be, to learn that they needed to scroll when browsing the web? **A user who can't learn to scroll within a few hours certainly won't be using the internet for very long.**

In Joel Spolsky's excellent [User Interface Design for Programmers](#), he notes **the difference between Usability and Learnability**:

“It takes several weeks to learn how to drive a car. For the first few hours behind the wheel, the average teenager will swerve around like crazy. They will pitch, weave, lurch, and sway. If the car has a stick shift they will stall the engine in the middle of busy intersections in a truly terrifying fashion.

“If you did a usability test of cars, you would be forced to conclude that they are simply unusable.

“This is a crucial distinction. When you sit somebody down in a typical usability test, you're really testing how learnable your interface is, not how usable it is. Learnability is important, but it's not everything. Learnable user interfaces may be extremely cumbersome to experienced users. If you make people walk through a fifteen-step wizard to print, people will be pleased the first time, less pleased the second time, and downright ornery by the fifth time they go through your rigamarole.

“Sometimes all you care about is learnability: for example, if you expect to have only occasional users. An information kiosk at a tourist attraction is a

good example; almost everybody who uses your interface will use it exactly once, so learnability is much more important than usability. But if you're creating a word processor for professional writers, well, now usability is more important.

“And that's why, when you press the brakes on your car, you don't get a little dialog popping up that says ‘Stop now? (yes/no).’”

I was greatly impressed with the [expanded book version](#) of Joel's [User Interface Design for Programmers page](#). I sort of assumed that the book was a mildly enhanced reprint of the HTML, but 7 of the 18 chapters are completely new material. Based on a few quick Google searches, they really are new. And the full color printing is fantastic!

I've read a bunch of UI books, and Joel's is easily in my top three. I'll definitely be adding it to my [recommended developer reading list](#).

OceanofPDF.com

Google's Number One UI Mistake

Google's [user interface minimalism](#) is admirable. But there's one part of their homepage UI, downloaded millions of times per day, that leaves me scratching my head:



Does anyone actually use the "I'm Feeling Lucky" button? I've been an avid Google user since 2000; I use it somewhere between dozens and hundreds of times per day. But I can count on one hand the number of times I've clicked on the ["I'm Feeling Lucky" button](#).

I understand this was a clever little joke in the early days of Google—hey, look at us, we're a search engine that actually works!—but is it really necessary to carry this clever little joke forward ten years and display it on the monitors of millions of web users every day? We get it already. **Google is awesomely effective.** That's why I use it so much. That's why Google is [the start page for the internet](#), loading the Google homepage is virtually [synonymous with internet access](#), and the verb "to Google" is at risk of becoming a [genericized trademark](#). Google has won so decisively, so utterly, and so completely that the power they now wield over the internet actually scares me a little. Okay, it scares me a lot.

So can we get rid of the superfluous button now?

You might say [it's only one more button](#), so where's the harm. I say **giving a feature that's used less than one percent of the time parity with the "Search" button is a needless distraction for users.** Furthermore, the "I'm Feeling Lucky" button is only available on the homepage—it's not a part of any browser toolbar searches, and Google's intermediate search page results don't offer it, either. Why not standardize and stick with the simple, single "Search" button that everyone understands and expects, on every page? Why muddy the waters with a button that's so rarely useful, and on the homepage of all places? The thought necessary to mentally omit this needless button from the page may be miniscule—but multiply that by the millions upon millions of users who are affected, and all of a sudden it starts to add up to real time. [Don't make us think!](#)

If you're an advanced computer user, you may be wondering why we bother with Search buttons at all when we have [a perfectly good ENTER key on our keyboards](#). As shocking as this may be to us [homo logicus](#), not everyone understands how that works. Sure, we think it's crazy to take our hand off the keyboard, where we were just typing our search query, move it all the way over to the mouse, then carefully move the mouse pointer to a button and left-click it... when we could just take that very same hand, already poised over the keyboard, and lazily tap the ENTER key.

But typical users [don't really understand basic keyboard shortcuts](#). They love their mice, and their big, fat, honking "Search" buttons. That's why the current versions of Firefox and IE both have **an integrated "go" button directly next to the address bar**—so users have something obvious to click once they've typed the URL into the address bar. Otherwise, I guess, they'd sit there wondering if their computer had frozen.



Personally, I always use the keyboard ENTER key to complete my searches, but I'd be open to a keyboard shortcut such as SHIFT+ENTER that invoked

the Lucky function. I still can't imagine using it more than once a week at most—and that's probably an optimistic estimate.

Strunk and White urged us to [Omit Needless Words](#):

“Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts. This requires not that the writer make all his sentences short, or that he avoid all detail and treat his subjects only in outline, but that every word tell.”

I urge us to **Omit Needless Buttons**. I hope the "I'm Feeling Lucky" button isn't considered [a sacred cow](#) at Google. Removing it would be one small step for Google, but a giant collective improvement in the default search user interface for users around the world.

OceanofPDF.com

But It's Just One More

The [Windows Live Local](#) mapping service is surprisingly difficult to use. It certainly looks simple enough:



Like everyone else, the first thing I do when I encounter a new mapping solution is try my current address. In this case, it's my work address. But when I press enter, I get this error:

“No results were found. Try another search, or if entering an address, enter it in the Where box. Click help to learn more.”

This is admittedly a sample size of one. But **everyone I know makes this mistake when using Windows Live Local search for the first time.** Yes, the two text boxes are labeled. Sort of. But [users won't read anything you put on the screen](#), even so-called professional computer users like ourselves. **There's simply one textbox too many on that form.**

It may seem irrational to declare that two of anything is one too many, but consider [these stopwatches](#):



Here's a stopwatch with one button. So this button must start, stop, and reset the time. It's a little overloaded, but like an [Apple mouse](#), at least nobody gets confused. In theory.



Let's add one more button. Maybe one button starts and stops, and the other resets? Or maybe one button starts and the other stops. But which one? It'll take a bit of trial and error to get this to work.



Now we add another button. And an extra sweeping hand. I don't even know where to begin. The complexity just went up exponentially.



This stopwatch has three colored buttons. And no sweeping hand. The colors definitely help: red means stop, green means go. So I'm guessing black is reset.

The last stopwatch illustrates that it is possible to add interface elements without adding confusion. But you have to do it very carefully. **If you have to add "just one more..." of any UI element, be sure that you're not adding the one UI element that breaks the camel's back.**

OceanofPDF.com

Just Say No

Derek Sivers relates an interesting [Steve Jobs anecdote](#):

“In June of 2003, Steve Jobs gave a small private presentation about the iTunes Music Store to some independent record label people. My favorite line of the day was when people kept raising their hand saying, ‘Does it do (x)?’, ‘Do you plan to add (y)?’. Finally Jobs said, ‘Wait wait—put your hands down. Listen: I know you have a thousand ideas for all the cool features iTunes could have. So do we. But we don't want a thousand features. That would be ugly. Innovation is not about saying yes to everything. It's about saying NO to all but the most crucial features.’”

I've worked on dozens of projects that have essentially killed themselves with kindness: piling on feature after feature trying to be all things to all users. This rarely ends well.

After a few years in the trenches, I think many software developers begin to internalize the **Just Say No** philosophy. Both extremes are dangerous, but I think **Yes To Everything** has a greater potential to fail the entire project. If you're going to err on either side, try to err on the side of simplicity. Keep a laser-like focus on doing a few things, and doing them exceptionally well.

It's easy to dismiss Just Say No as a negative mindset, but I think it is a healthy and natural reaction to the observation that [optimism is an occupational hazard of programming](#). It takes a lot more courage to say "no" than it does to nod along in the hopes of pleasing everyone.

The implicit lesson is not to literally say no to everything—but to weigh very carefully the things you are doing. For a very interesting case study, check out Google Blogosoped's [Illustrated Chronicles of the Portal Plague](#).

OceanofPDF.com

UI Is Hard

Some users [commenting](#) on the poor pre-game user interface in EA's [Battlefield 2](#):

Poster #1: They need to stop hiring angry little men and romantically spurned women to design user interfaces.

Poster #2: But doesn't that describe most programmers?

Poster #3: No, that describes all programmers.

It's funny because it's true. Not the romantically spurned part, mind you, but the accusation that most programmers are bad at designing user interfaces. That's partly because [UI is hard](#):

“GUI builders make GUI programming look easy. Nearly anybody can whip up a decent-looking GUI in no time at all using a GUI builder. Done.

“It is much harder to whip up a quick and dirty EJB system, giving the impression that server-side coding is harder to do. A bad programmer will continue to struggle with EJB, but a good programmer will find ways to automate nearly every aspect of EJB. That's the secret of server-side programming: it is very well-defined and repetitive. Thus, it can be automated.

“Take your favorite Model-Driven-Architecture (MDA) tool. They work best when generating server-side code, things like EJBs, database access code, and web services. They might be able to generate a rudimentary GUI, but a really GREAT GUI cannot be automated.”

But programmers are partly to blame, too. Most programmers [begin by thinking about the code instead of the user interface](#):

“John almost hit on the most important point in all of this. No one else did. 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 (Terminal window for you Mac OS X users). 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.

“The second problem is that the tools we use to create UI are often good tools for more simple usability issues, but tend to fall well short when it comes to designing UI for a more complex set of user scenarios. Forms designers are great when you're working within the problem domain that forms are intended to solve, but once you step outside those problem domains, the work gets much harder. Use a more flexible tool, like Xcode's nib tool and the Mac OS X HView object, and you're going to have to write considerably more code just to manage the UI objects.”

This is also known as [UI First Development](#), but I can't find many other references.

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

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

OceanofPDF.com

IV.

Testing

OceanofPDF.com

Good Test/Bad Test

After years of building ad-hoc test harnesses, I finally adopted formal unit testing on a recent project of mine using [NUnit](#) and [TestRunner](#). It was gratifyingly simple to get my first unit tests up and running:

```
<TestFixture(> _
Public Class UnitTests

    Private _TargetString As String
    Private _TargetData As Encryption.Data

    <TestFixtureSetUp(> _
    Public Sub Setup()
        _TargetString = "an enigma wrapped in a mystery slathered in secret sauce"
        _TargetData = New Encryption.Data(_TargetString)
    End Sub

    <Test(), Category("Symmetric")> _
    Public Sub MyTest()
        Dim s As New Encryption.Symmetric(Encryption.Symmetric.Providers.DES)
        Dim encryptedData As Encryption.Data
        Dim decryptedData As Encryption.Data

        encryptedData = s.Encrypt(_TargetData)
        decryptedData = s.Decrypt(encryptedData)

        Assert.AreEqual(_TargetString, decryptedData.ToString)
    End Sub

End Class
```

It's a great system because I can tell what it does and how it works just by looking at it. You can't knock simplicity. The problem with unit testing, then, is not the implementation. It's determining what to test. And how to test it. Or, more philosophically, **what makes a good test?**

You'll get no argument from me on the fundamental value of unit testing. Even the most trivially basic unit test, as shown in the code sample above, is a huge step up from the testing most developers perform—which is to say, **most developers don't test at all!** They key in a few values at random and click a few buttons. If they don't get any unhandled exceptions, that code is ready for QA!

The real value of unit testing is that **it forces you to stop and think about testing.** Instead of a willy-nilly ad-hoc process, it becomes a series of hard, unavoidable questions about the code you've just written:

- How do I test this?
- What kinds of tests should I run?
- What is the common, expected case?
- What are some possible unusual cases?
- How many external dependencies do I have?
- What system failures could I reasonably encounter here?

Unit tests don't guarantee correct functioning of a program. I think it's unreasonable to expect them to. But writing unit tests does guarantee that the developer has considered, however briefly, these truly difficult testing questions. And that's clearly a step in the right direction.

One of the other things that struck me about unit testing was the challenge of balancing unit testing with the massive refactoring all of my projects tend to go through in their early stages of development. And, [as Unicode Andy points out](#), I'm not the only developer with this concern:

“My main problem at the moment with unit tests is when I change a design I get a stack of failing tests. This means I'm either going to write less tests or make fewer big design changes. Both of which are bad things.”

To avoid this problem, I'm tempted to take the old-school position that tests should be coded later rather than sooner, which runs counter to the hippest theories of [test-first development](#). How do you balance the need to write unit tests with the need to aggressively refactor your code? Does test-first reduce the refactoring burden, or do you add unit tests after your design has solidified?

Unit Testing vs. 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 off 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.

OceanofPDF.com

Sometimes It's a Hardware Problem

One of our best servers at work was inherited from a previous engagement for x64 testing: it's a dual Opteron 250 with 8 gigabytes of RAM. Even after a year of service, those are still decent specs. And it has a nice upgrade path, too: the [Tyan Thunder K8W](#) motherboard it's based on supports up to 16 gigabytes of memory, and [the latest dual core Opterons](#).

Anyway, we have it set-up for [Virtual Server 2005 R2 duties](#), running Windows Server 2003 x64. However, there was some anomalous behavior:

- Virtual Server reported weird error messages: "Some nodes of this machine do not have local memory. This can cause virtual machines to run with degraded performance."
- The machine spontaneously rebooted during the day and overnight.

We've used this server for over a year and never experienced anything problematic with it. The weirdness only started with the server's new role.

The first thing we did was **update the BIOS to the latest version, and make sure we had all the latest x64 chipset and platform drivers installed**. This is always a good first troubleshooting step—it's the hardware equivalent of taking two aspirins and calling in the morning. This resolved the "some nodes of this machine do not have local memory" error. However, the machine still spontaneously rebooted overnight, even with the latest BIOS and drivers.

At this point I began to suspect a hardware problem. Troubleshooting hardware stability can be difficult. But **you can troubleshoot hardware stability quite effectively with the right software: [Memtest86+](#) and [Prime95](#)**.

1. Testing memory stability with [Memtest86+](#)

We started with Memtest86+ because we already suspected the memory. Memtest86+ isn't the only memory testing diagnostic out there, but it's

probably the most well-known. Microsoft also offers their [Windows Memory Diagnostic](#) utility, which works exactly the same way. Memtest86+ is [available in several forms from the Memtest86+ web site](#). We chose the ISO image, which we burned to CD. Boot from the Memtest86 CD, and it'll kick off the test run.

```
Memtest86 v1.65 | Pass 8% ###
Athlon 64 X2 1048 MHz | Test 1%
L1 Cache: 128K 1652MB/s | Test #4 [Moving inversions, random pattern]
L2 Cache: 1024K | Testing: 108K - 192M 192M
Memory : 192M 835MB/s | Pattern: 45a2d44d
Chipset : Intel i440BX

WallTime  Cached  RsdMem  MemMap  Cache  ECC  Test  Pass  Errors  ECC Errs
-----
0:00:48  192M    132K  e820-Std  on  off  Std    0    0

(ESC)Reboot (c)configuration (SP)scroll_lock (CR)scroll_unlock
```

It took about 30-45 minutes to test 4 gigabytes of memory. The progress bar at the top right gives you an indication of how long the test has to run; there are 8 total tests in the standard test run. Beware, because it'll start repeating at test #1 after the first pass!

2. Testing CPU stability with [Prime95](#)

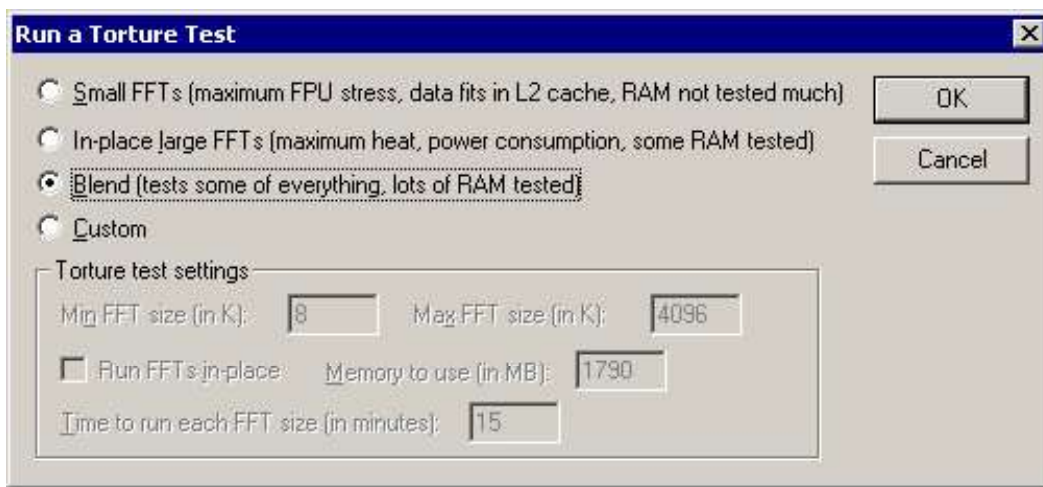
Prime95 is my single favorite PC stability testing tool. If your PC can't pass an overnight Prime95 run, it absolutely, positively has a hardware problem. (CPUs are almost never defective; it's usually a heat or power supply related failure.) Although Prime95 is primarily a CPU test, it can also be a pretty good memory test, too. After downloading it, go to the Options menu and select Torture Test.

If you want to test CPU stability exclusively, choose "Small FFTs."

If you want to test for CPU stability and memory stability, choose "Blend."

If you have a Dual (or Quad) CPU machine, you must run multiple instances of Prime95 to load each CPU. The easiest way to do this is to copy the Prime95 folder and run multiple executables, each one from a unique folder. You may want to set CPU affinity on the executables with Task Manager, but the scheduler will take care of loading all the CPUs just fine by itself.

A bit of warning, though: when Prime95 says "lots of RAM tested," they mean it. We tried running two instances of "Blend" with only 4 gigabytes of memory installed on the server and we nearly crushed the pagefile; both instances allocated nearly 6 gigabytes!



In my experience, Prime95 will error out almost immediately if your CPUs or memory are unstable. This is great for troubleshooting because you know quickly if there's a problem or not. If you can run Prime95 "small FFTs" for an hour, it's highly likely that the CPU isn't your problem. And if you can run the same test overnight, CPU problems can be definitively ruled out.

In the case of our wayward server, Memtest86+ showed us rare, intermittent memory problems. But Prime95 consistently failed almost immediately when running the "blend" test. When we switched Prime95 to "small FFTs," it ran two instances for an hour just fine. Clearly a memory issue! Using a combination of Memtest86+ and Prime95, we found that **our server was totally stable with 4 gigabytes of memory installed**; the minute we put in all 8 gigabytes, we couldn't pass one or both tests.

Since 8 gigabytes of memory is essential for a VM server, removing memory wasn't an option. On a hunch, I switched the memory speed from

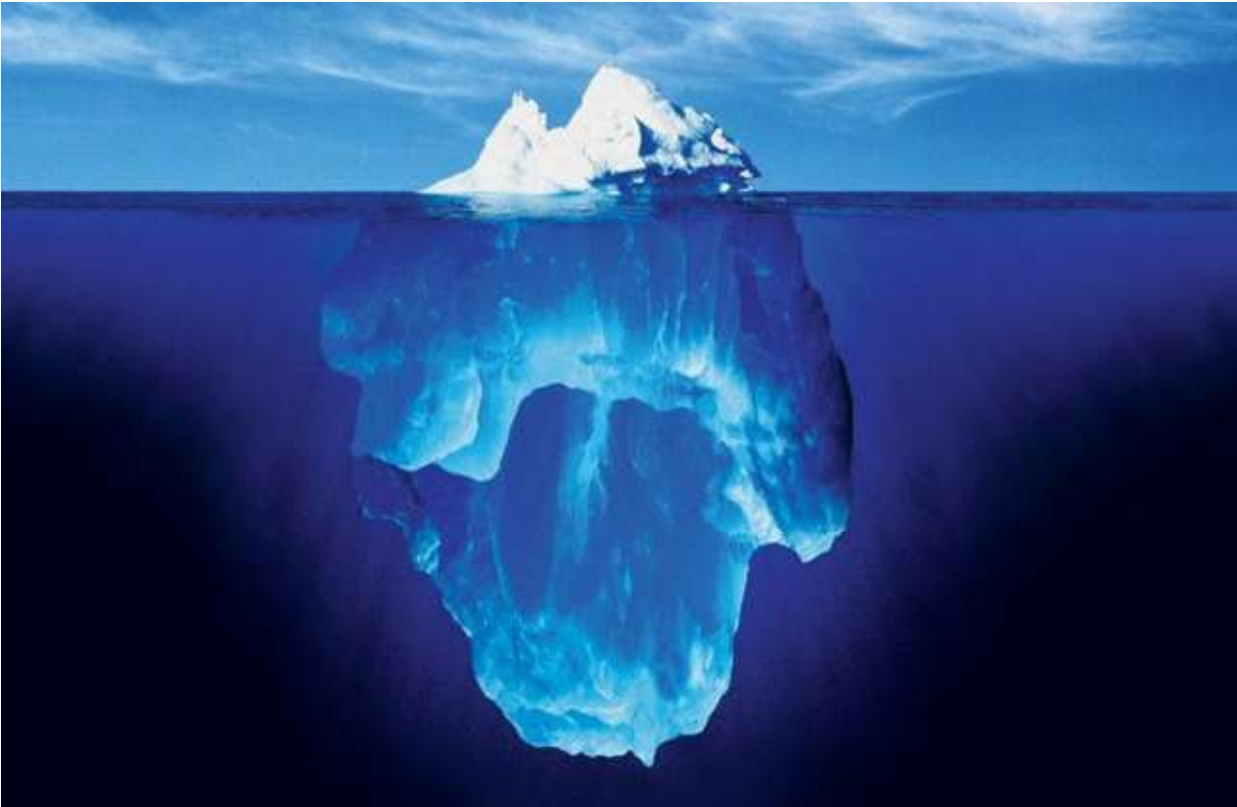
200 MHz to 166 MHz in the BIOS. Now both Prime95 blend and Memtest86+ pass without incident.

Although software is notoriously unreliable, we can't always blame the software. **Sometimes you really do have a hardware problem.**

OceanofPDF.com

Exception-Driven Development

If you're waiting around for **users to tell you about problems with your website or application**, you're only seeing a tiny fraction of all the problems that are actually occurring. The proverbial tip of the iceberg.



Also, if this is the case, I'm sorry to be the one to have to tell you this, but you kind of suck at your job—which is to **know more about your application's health than your users do**. When a user informs me about a bona fide error they've experienced with my software, I am deeply embarrassed. And more than a little ashamed. I have failed to see and address the issue before they got around to telling me. I have neglected to [crash responsibly](#).

The first thing any responsibly run software project should build is an **exception and error reporting facility**. Ned Batchelder likens this to [putting an oxygen mask on yourself before you put one on your child](#):

“When a problem occurs in your application, always check first that the error was handled appropriately. If it wasn't, always fix the handling code first. There are a few reasons for insisting on this order of work:

1. With the original error in place, you have a perfect test case for the bug in your error handling code. Once you fix the original problem, how will you test the error handling? Remember, one of the reasons there was a bug there in the first place is that it is hard to test it.
2. Once the original problem is fixed, the urgency for fixing the error handling code is gone. You can say you'll get to it, but what's the rush? You'll be like the guy with the leaky roof. When it's raining, he can't fix it because it's raining out, and when it isn't raining, there's no leak!”

You need to have a central place that all your errors are aggregated, a place that all the developers on your team know intimately and visit every day. On Stack Overflow, we use a custom fork of [ELMAH](#).



The screenshot shows a web browser window with the title "Error Log for /LM/W3SVC/4/ROO...". The main heading is "8 Errors; last 44 sec ago". Below this is a table with columns for Type, Error, Url, and Time. The table contains 8 rows of error data. The first four rows are "Argument" errors for "messagetypeid" on the "/messages/mark-as-read" URL. The next three rows are "NullReference" errors for "Object reference not set to an instance of an object." on the "/users/authenticate" URL. The final row is a "Validation" error for "Email has an invalid value: does not appear to be valid" on the "/users/authenticate" URL. At the bottom, it says "Server time is 02/11/2009 01:00:39 PST".

| Type | Error | Url | Time |
|---------------|---|---|-------------------|
| Argument | messagetypeid X | /messages/mark-as-read | 09/02/11 00:59:55 |
| Argument | messagetypeid X | /messages/mark-as-read | 09/02/11 00:50:21 |
| Argument | messagetypeid X | /messages/mark-as-read | 09/02/11 00:49:44 |
| Argument | messagetypeid X | /messages/mark-as-read | 09/02/11 00:40:51 |
| NullReference | Object reference not set to an instance of an object. X | /users/authenticate | 09/02/10 22:26:43 |
| NullReference | Object reference not set to an instance of an object. X | /users/authenticate | 09/02/10 22:26:05 |
| NullReference | Object reference not set to an instance of an object. X | /users/authenticate | 09/02/10 22:25:32 |
| Validation | Email has an invalid value: does not appear to be valid X | /users/authenticate | 09/02/10 21:44:37 |

Server time is 02/11/2009 01:00:39 PST

We monitor these exception logs daily; sometimes hourly. **Our exception logs are a de-facto do list for our team.** And for good reason. Microsoft has collected similar sorts of failure logs for years, both for themselves and other software vendors, under the banner of their Windows Error Reporting service. The [resulting data](#) is compelling:

“When an end user experiences a crash, they are shown a dialog box which asks them if they want to send an error report. If they choose to send the

report, WER collects information on both the application and the module involved in the crash, and sends it over a secure server to Microsoft.

“The mapped vendor of a bucket can then [access the data for their products](#), analyze it to locate the source of the problem, and provide solutions both through the end user error dialog boxes and by providing updated files on Windows Update.

“Broad-based trend analysis of error reporting data shows that 80 percent of customer issues can be solved by fixing 20 percent of the top-reported bugs. Even addressing one percent of the top bugs would address 50 percent of the customer issues. The same analysis results are generally true on a company-by-company basis too.”

Although [I remain a fan of test driven development](#), the speculative nature of the time investment is one problem I've always had with it. **If you fix a bug that no actual user will ever encounter, what have you actually fixed?** While there are [many other valid reasons to practice TDD](#), as a pure bug fixing mechanism it's always seemed far too much like premature optimization for my tastes. I'd much rather spend my time fixing bugs that are problems in practice rather than theory.

You can certainly do both. But given a limited pool of developer time, I'd prefer to allocate it toward fixing problems real users are having with my software based on cold, hard data. That's what I call **Exception-Driven Development**. Ship your software, get as many users in front of it as possible, and intently study the error logs they generate. Use those exception logs to hone in on and focus on the problem areas of your code. Rearchitect and refactor your code so the top 3 errors can't happen any more. [Iterate rapidly](#), deploy, and repeat the process. This data-driven feedback loop is so powerful you'll have (at least from the users' perspective) a rock stable app in a handful of iterations.

Exception logs are possibly the most powerful form of feedback your customers can give you. It's feedback based on shipping software that you don't have to ask or cajole users to give you. Nor do you have to interpret your users' weird, semi-coherent ramblings about what the problems are. The actual problems, with stack traces and dumps, are collected for you, automatically and silently. **Exception logs are the ultimate in customer feedback.**

@lazycoder getting real feedback from customers by shipping is more valuable than any amount of talking to or about them beforehand

about 6 hours ago from twitterrific in reply to lazycoder



Carnage4Life

Dare Obasanjo

Am I advocating shipping buggy code? Incomplete code? Bad code? Of course not. I'm saying that the sooner you can get your code out of your editor and in front of real users, the more data you'll have to improve your software. Exception logs are a big part of that; so is usage data. And you should talk to your users, too. If you can bear to.

Your software will ship with bugs anyway. [Everyone's software does](#). Real software crashes. Real software loses data. Real software is hard to learn, and hard to use. The question isn't how many bugs you will ship with, but **how fast can you fix those bugs?** If your team has been practicing exception-driven development all along, the answer is—why, we can improve our software in no time at all! Just watch us make it better!

And that is sweet, sweet music to every user's ears.

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

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

OceanofPDF.com

V.

Know Your User

OceanofPDF.com

The Rise and Fall of Homo Logicus

Of all the professional hubris I've observed in software developers, perhaps the greatest sin of all is that **we consider ourselves typical users**. We use the computer obsessively, we know a lot about how it works, we even give advice to friends and relatives. We are experts. Who could possibly design software better than us superusers? What most developers don't realize is how freakishly outside the norm we are. We're not even remotely average—we are the edge conditions. I've often told program managers: [if you are letting me design your software, your project is in trouble.](#)

In [The Inmates Are Running the Asylum](#), Alan Cooper labels this phenomenon Homo Logicus:

“Homo logicus desires to have control over things that interest them, and the things that interest them are complex, deterministic systems. People are complex, but they don't behave in a logical and predictable way, like machinery. The best machinery is digital, because it can be the most complex, sophisticated, and easily changed by the programmer.

The price of control is always more effort and increased complexity. Most people are willing to make a moderate effort, but what differentiates programmers from most people is their willingness and ability to master extreme complexity. It is a satisfying part of the programmer's job to know and manage systems composed of many interacting forces. Flying airplanes is the archetypal programmer's avocation. The cockpit control panel of an airplane is packed with gauges, knobs, and levers, but programmers thrive on those daunting complexities. Homo logicus finds it fun and engaging, despite (because of!) the months of rigorous study required. Homo sapiens would rather ride along as passengers.

For Homo logicus, control is their goal and complexity is the price they will pay for it. For normal humans, simplicity is their goal, and relinquishing control is the price they will pay. In software-based products, control translates into features. For example, in Windows 95, the "Find File" function gives me lots of control over the procedure. I can specify which area of my disk to search, the type of file to search for, whether to search by

file name or by file contents, and several other parameters. From a programmer's point of view, this is very cool. For some extra up-front effort and understanding, he gets to make the search faster and more efficient. Conversely, the user's point of view is less rosy because he has to specify the area of the search, the type of file to search for, and whether to search by name or contents. Homo sapiens would gladly sacrifice the odd extra minute of compute time if they didn't have to know how the search function works. To them, each search parameter is just another opportunity to enter something incorrectly. The probability of making a mistake and the search function failing is higher, not lower, with the added flexibility. They would gladly sacrifice all that unnecessary complexity, control, and understanding in order to make their job simpler.

Homo logicus are driven by an irresistible desire to understand how things work. By contrast, Homo sapiens have a strong desire for success. While programmers also want to succeed, they will frequently accept failure as the price to pay for understanding. There's an old joke about engineers that gives some insight into this need to understand.

Three people are scheduled for execution: a priest, an attorney, and an engineer. First, the priest steps up to the gallows. The executioner pulls the lever to drop the hatch, but nothing happens. The priest claims divine intervention and demands his release, so he is set free. Next, the attorney takes a stand at the gallows. The executioner pulls the lever, but again nothing happens. The attorney claims another attempt would be double jeopardy and demands release, so he is set free. Finally, the engineer steps up to the gallows, and begins a careful examination of the scaffold. Before the executioner can pull the lever, he looks up and declares, 'Aha, here's your problem.'

Cooper goes on to list a few more traits of Homo Logicus:

- trades simplicity for control
- exchanges success for understanding
- focuses on what is possible to the exclusion of what is probable
- acts like a jock

Pity the poor user, merely a Homo Sapiens, who isn't interested in computers or complexity; he just wants to get his job done.

Anybody can build a complex application that nobody can figure out how to use. That's easy. Building an application that's simple to use.. well, now that takes actual skill. I'm not sure you need high priced [interaction designers](#) to achieve this goal, but **you do have to stop thinking like Homo Logicus—and start thinking like Homo Sapiens.**

OceanofPDF.com

Ivory Tower Development

I've always discouraged **ivory tower development**—teams where developers are cloistered away for years in their high towers, working on technical software wizardry. These developers have no idea how users will respond to their software they're creating. They probably couldn't even tell you the last time they met a user! **In the absence of any other compelling evidence, developers assume everyone else is a developer.** I hope I don't have to tell you how dangerous that is.

In my experience, the more isolated the developers, the worse the resulting end product. It doesn't help that most teams have a layer of business analysts who feel it is their job to shield developers from users, and vice-versa. It's dangerous to create an environment where [developers have no idea who the users are](#):

“I gave a presentation to an all-hands meeting for a division of Sun, and I asked the group to raise their hands if they'd met a live customer in the last 30 days. Couple of hands went up. ‘The last 90 days?’ One more. ‘The last year?’ Another two. There were over 100 people in that room directly responsible for deliverables that went straight to users... in this case, Java training courses.

“This flies in the face of some software development models that believe if you've done your specifications right, there should be no need for the ‘workers’ (programmers, writers, etc.) to ever come in contact with real users. That's nonsense. What users are able to articulate before they have something is rarely a perfect match for what they say after they've actually experienced it. It's just like market research: people can't tell you in advance exactly how they'll react to something. They just have to try it. And you have to be there to watch. And listen. And learn. And then take what you learned and go back and refine. Which is why the old waterfall model is pretty much the worst thing to ever happen to users.”

Make the effort to expose your developers to users throughout your project lifecycle. Bring one developer from your team to every meeting

with users. Involve developers in your usability and acceptance testing. Nothing removes a developer's [Homo Logicus](#) blinders faster than seeing a typical user struggle with basic computer applications. Developers simply cannot comprehend that the average user doesn't even know what ALT+TAB does, much less how to use it. They have to see it to believe it.

Most projects I work on these days are internal. I define internal projects as projects where **users are forced to use your application whether they want to or not**. So much for free will. And, too many times, so much for concerns about software quality. As Joel says: [sadly, lots of internal software sucks pretty badly](#). It's true. And it is sad. This is another form of Ivory Tower Development: what incentive do I have to care about the concerns of our "customer" when their job requires them to use my application?

I'd much rather work on projects with paying customers, or at least treat internal projects as if users were paying real money for your product. That engenders what [Eric Sink](#) calls a [mutual trust relationship](#):

“When people buy software from you, they expect a lot, both now and in the future:

- They trust that your product will work on their machines.
- They trust that you will help them if they have problems.
- They trust that you will continue to improve the product.
- They trust that you will provide them with a reasonable and fairly priced way of getting those improved versions.
- They trust that you are not going out of business anytime soon.

So, by asking users to pay for your software, you are asking them to trust you. But how much do you trust them?

The vendor/user relationship is like a relationship between two people. And relationships don't work without mutual trust. If one side expects trust but is unwilling to give it, the relationship will fail. So often I see software entrepreneurs who don't want to trust their users at all. It is true that trusting someone makes us vulnerable. Just as in a human relationship, trust is a

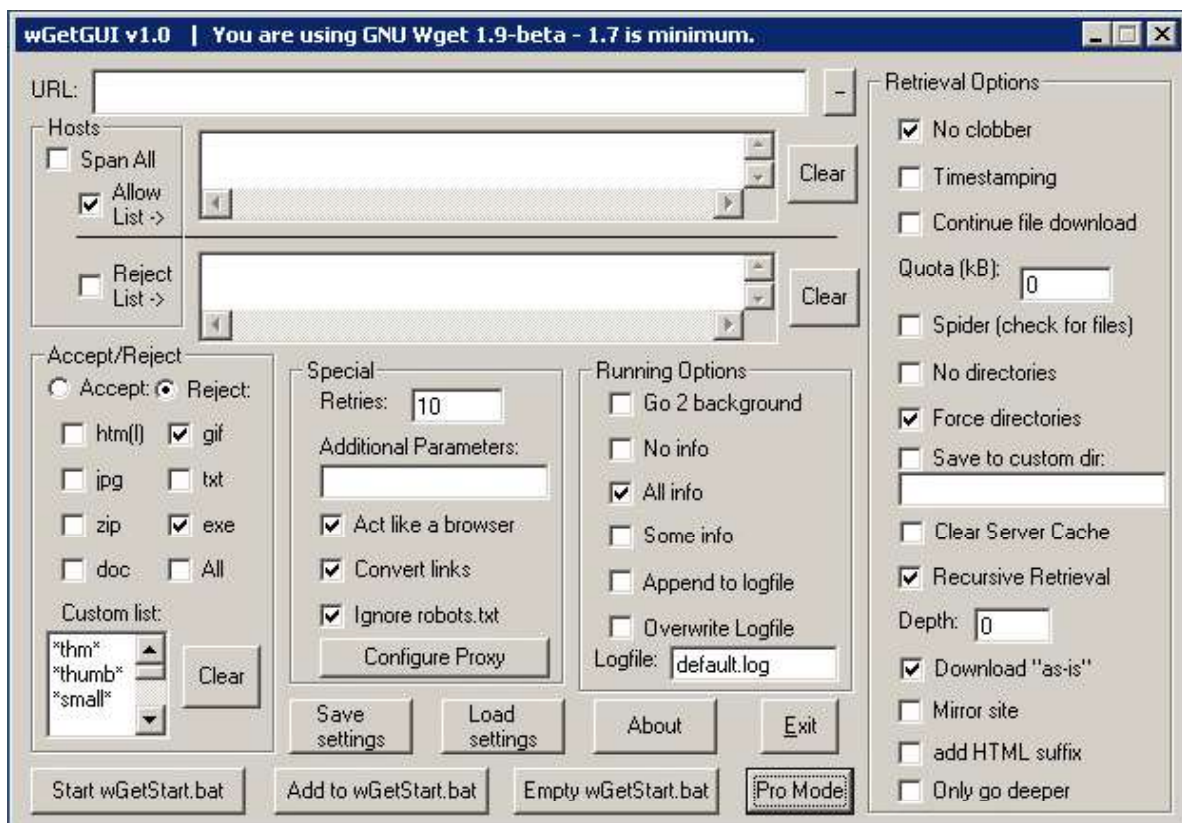
risk. We might get hurt. But without that trust, the relationship isn't going to work at all.”

I've actually begun to think that **internal departments [of large companies] should act as micro-ISVs**, charging their users for the applications they build, and actively marketing and selling them to other groups inside the organization. I think that would lead to a leaner, meaner, and ultimately more healthy organization. Plus, the [boondoggle](#) projects so common at large companies would die naturally due to lack of demand.

OceanofPDF.com

This Is What Happens When You Let Developers Create UI

Deep down inside every software developer, **there's a budding graphic designer waiting to get out.** And if you let that happen, you're in trouble. Or at least your users will be, anyway:



Joseph Cooney calls this [The Dialog](#):

“A developer needed a screen for something, one or two text boxes and not much more, so they created ‘the dialog’, maybe just to ‘try something out’ and always with the intention of removing it before the product ships. They discovered they needed a few more parameters, so a couple more controls were added in a fairly haphazard fashion. ‘The dialog’ exposes ‘the feature,’ something cool or quite useful. Admittedly ‘the feature’ is more tailored towards power users, but it's still pretty cool. The developer thinks

of new parameters that would make ‘the feature’ even more powerful and so adds them to the dialog. Maybe a few other developers or power users see ‘the dialog’ and also like ‘the feature.’ But why doesn't it expose *this* parameter? New controls are added. Pretty soon the technical team are so used to seeing ‘the dialog’ the way it is that they become blind to its strange appearance. Ship time approaches and the product goes through more thorough testing, and ‘the dialog’ is discovered, but it is too late to be heavily re-worked. Instead it is given a cursory spruce-up.”

If you let your developers create your UI, hilarity ensues, as in [this classic OK/Cancel strip](#). But when [The FileMatrix](#) is unleashed upon unsuspecting users, it's more like a horror movie. I still get chills. And like a bad horror movie franchise, [the FileMatrix is still alive and kicking](#), folks.

Friends don't let friends produce Developer UI.

Part of being a good software developer is knowing your limits. Either [copy something that's already well designed](#), or have the good sense to stick to coding and leave the graphic design to the experts.

OceanofPDF.com

Defending Perpetual Intermediacy

How many things would you classify yourself as "expert" at? I drive to and from work every day, but I hardly consider myself an expert driver. I brush my teeth at least twice every day, and I'm no expert on oral care; just ask my dentist. I use Visual SourceSafe all the time, but I rarely use the more esoteric branching, pinning, and rollback features. I have to look through the help files when I do those things. I am a **perpetual intermediate** at a vast array of tasks, and expert at only a very, very tiny number of tasks. In "[The Inmates are Running the Asylum](#)," Alan Cooper makes a similar case for users as perpetual intermediates:

“The experience of people using interactive systems—as in most things—tends to follow the classic bell curve of statistical distribution. For any silicon-based product, if we graph the number of users against their particular skill level, there will be a few beginners on the left side, a few experts on the right, and a preponderance of intermediate users in the center.

“But statistics don't tell the whole story. This is a snapshot frozen in time, and while most people—the intermediates—tend to stay in that category for a long time, the people on the extreme ends of the curve—the beginners and experts—are always changing. The difficulty of maintaining a high level of expertise means that experts come and go rapidly. Beginners, on the left side of the curve, change even more rapidly.

“Although everybody spends some minimum time as a beginner, nobody remains in that state for long. That's because nobody likes to be a beginner, and it is never a goal. People don't like to be incompetent, and beginners—by definition—are incompetent. Conversely, learning and improving is natural, rewarding, and lots of fun, so beginners become intermediates very quickly. For example, it's fun to learn tennis, but those first few hours or days, when you can't return shots and are hitting balls over the fence are frustrating. After you have learned basic racket control, and aren't spending all of your time chasing lost balls, you really move forward. That state of beginnerhood is plainly not fun to be in, and everybody quickly passes

through it to some semblance of intermediate adequacy. If, after a few days, you still find yourself whacking balls around the tennis court at random, you will abandon tennis and take up fly-fishing or stamp collecting.

“The occupants of the beginner end of the curve will either migrate into the center bulge of intermediates, or they will drop off of the graph altogether and find some activity in which they can migrate into intermediacy. However, the population of the graph's center is very stable. When people achieve an adequate level of experience and ability, they generally stay there forever. Particularly with high cognitive friction products, users take no joy in learning about them. So they learn just the minimum and then stop. Only [Homo Logicus](#) finds learning about complex systems to be fun.”

Cooper goes on to decry the way software development is traditionally driven by opposite ends of the spectrum—developers as advocates for expert users, and marketing as advocates for beginners (which is typically their audience). Who speaks for the intermediate users?

I'll take this a bit further: **I think intermediate users are the only users that matter.** The huge body of intermediate users is so dominant that you can and should ignore both beginner and expert users. Developing software to accommodate the small beginner and expert groups consumes too much time and ultimately makes your application worse at the expense of your core user base—the intermediates. Beginners should either become intermediates or, in a manner of speaking, die trying. As for software targeting expert users exclusively (aka, developers), that's a tiny niche deserving of an entirely different design approach.

In my opinion, one of the most powerful tools we have for targeting intermediate users is the [Inductive User Interface](#). IUI, as a concept, is actually quite simple: take the best design elements of the web..

- Back button
- Single-click hyperlink navigation
- Activity-centric "everything on one page" model

and combine those with the best design elements of traditional GUIs..

- Rich interface
- High performance
- Leverages client resources (disk, memory, visuals)

The first major application to utilize IUI was [Microsoft Money 2000](#). My wife uses Money, and I distinctly remember installing Money 2000, and being absolutely blown away by how effective the UI was:

“The IUI model was developed during the creation of Microsoft Money 2000, an application for managing personal finances. Money 2000 is the product's eighth major release. Money 2000 is a large Microsoft Windows program with well over one million lines of code. Money 2000 is a Web-style application. It is not a Web site, but shares many attributes with Web sites. Its user interface consists of full-screen pages shown in a shared frame, with tools for moving back and forward through a navigational stack. On this foundation, Money 2000 adds a set of new user interface conventions that create a more structured user experience.”

The [Inductive User Interface](#) design is nothing more than good programming in practice: **never write what you can steal**. And stealing the wildly successful web UI metaphors is such an utter no-brainer. The only question I have is why it's taking so long.

We have bits and pieces of IUI in Windows XP (try Control Panel, User Accounts), and there's a lot of evidence that Microsoft [plans to utilize IUI much more heavily in Longhorn](#). But we don't have to wait for Longhorn; as responsible .NET developers, we should be building IUI interfaces today—as in this [MSDN Windows Forms sample](#).

Every User Lies

Heidi Adkisson notes that [features sell products](#), but the people buying those products often **don't use the very features they bought the product for in the first place.**

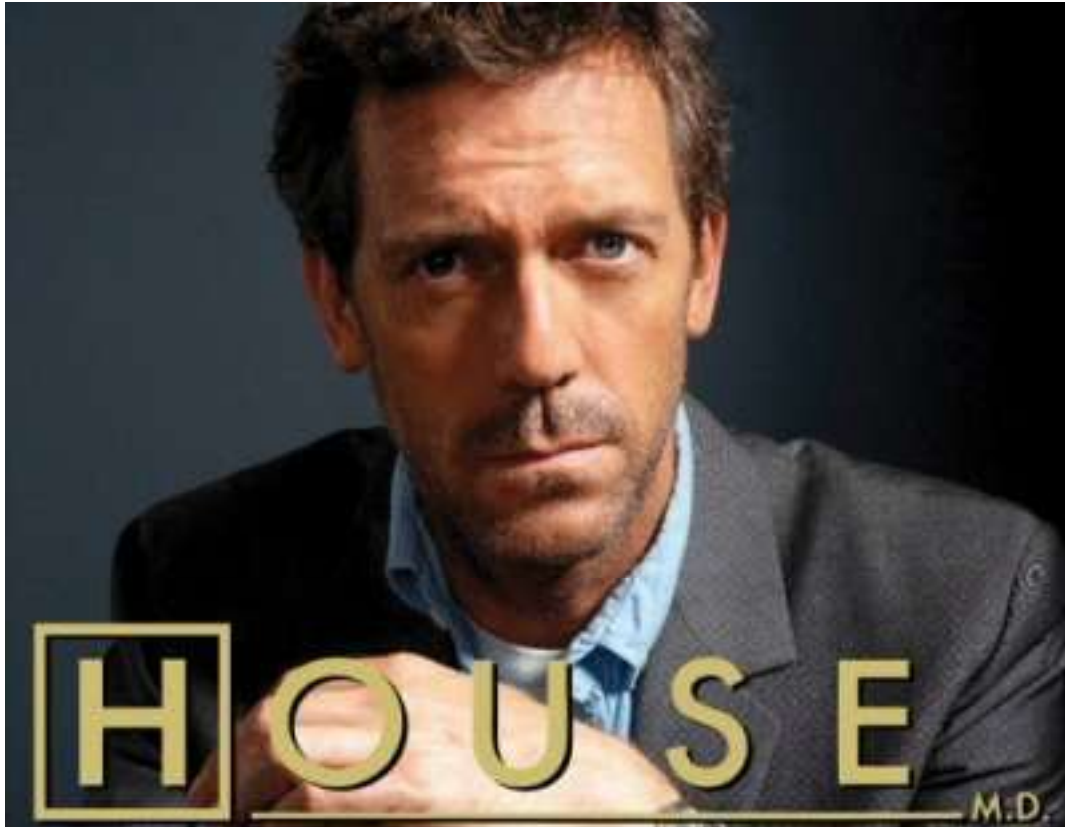
“A few years ago I did an extensive in-home study observing use of a particular computer hardware peripheral. Most people had high-end models with many features. But in my observation of use, only one ‘power user’ went beyond using anything but the core, basic features. These people had paid a premium for features they didn't use. However, when describing their purchase experience, it was clear *they aspired to using these features* and sincerely intended to. But, once the product was out of the box, the paradox of the active user took over. They never invested even the smallest amount of time to go beyond the basics (even though the extra features could have saved them time).

“In my experience it's people's *aspiration for the experience* that drives purchasing decisions. Ultimately, their aspirations may be different than the reality.”

It's interesting that Heidi used the hedge phrase "may be different" when **her own study data showed that users' aspirations and reality were almost always different.** Maybe she aspires to live in a world where aspirations and reality aren't so wildly divergent. I don't blame her. It'd be nice.

That disparity is why it's so important to [observe how users actually behave](#) versus the way they tell you they behave. People who do this professionally are called "economists." Observation is a powerful skill, and so is learning to disregard what people tell you in favor of judging them by their actions. No actions are more carefully considered than those that result in money flowing out of your pocket. That's why you owe it to yourself to read books like “[Freakonomics](#),” and maybe even [The Economist](#) magazine. It's also why the [Freakonomics blog](#) should be a part of your regular reading routine if it isn't already.

People lie not because they're all evil liars (although a few inevitably will be), but because they're usually lying to themselves in some way. Some lies are useful. Small social "white" lies grease the skids of social reality. Penetrating this veil of lies and intentions is one of the central themes of the excellent television show [House, M.D.](#):



The show plays up [subtle connections between](#) the House character and [Sherlock Holmes](#), which is appropriate, because it's very much a detective show at heart. The character Gregory House, as played by the brilliant [Hugh Laurie](#), is fond of stating "**Everybody lies.**" Parsing through all the irrational human behavior, and the inevitable lies—white or otherwise—makes for a gripping detective story indeed when lives are at stake.

Heidi referenced the [Paradox of the Active User](#), which has been around as a concept since 1987. I highly recommend reading the original paper, but if you don't have time, Jakob Nielsen [summarizes](#):

“Users never read manuals but start using the software immediately. They are motivated to get started and to get their immediate task done: they don't

care about the system as such and don't want to spend time up front on getting established, set up, or going through learning packages.

“The ‘paradox of the active user’ is a paradox because users would *save* time in the long term by learning more about the system. But that's not how people behave in the real world, so we cannot allow engineers to build products for an idealized rational user when real humans are irrational. We must design for the way users actually behave.”

There are a bunch of ways to restate the paradox of the active user. Cooper calls it [perpetual intermediacy](#). I think the easiest way to explain it is this: **every user lies**. Instead of asking users if they love your software—of course they love your software, it'd be rude to tell the people responsible just how mind-numbingly awful it really is—do what Gregory House does. [Observe whether or not they use the software, and how they use it](#). Rely on that behavioral data to design your software, and not the lies of your users, however well intentioned they may be.

OceanofPDF.com

Shipping Isn't Enough

Part of [Chuck Jazdzewski's fatherly advice to new programmers](#) is this nugget:

“Programming is fun. It is the joy of discovery. It is the joy of creation. It is the joy of accomplishment. It is the joy of learning. It is fun to see your handiwork displaying on the screen. It is fun to have your co-workers marvel at your code. It is fun to have people use your work. It is fun have your product lauded in public, used by neighbors, and discussed in the press. Programming should be fun and if it isn't, figure out what is making it not fun and fix it. However, shipping isn't fun. I often have said that shipping a product feels good, like when someone stops hitting you. Your job is completing the product, fixing the bugs, and shipping. If bugs need fixing, fix them. If documentation needs writing, write it. If code needs testing, test it. All of this is part of shipping. You don't get paid to program, you get paid to ship. Be good at your job.”

It's true. One key measure of success for any programmer is how much code you've shipped. **But merely shipping is not enough.** A more meaningful measure of success is to ask yourself how much code you've shipped to living, breathing, real-world users. But then [total users doesn't equal total usage](#), either.

How many users actually use your application? Now that's the ultimate metric of success.

But it's a little scary when you start doing the math. Rich Skrenta [explains](#):

“I was just an engineer in this group, but the reality of what was happening in the market to our product line started to seep in. Here I was putting all of this effort into stuff that never would be used by anyone. It was still intellectually challenging...like doing crossword puzzles or something. But it had no utility to the world.

“I started to look around and I saw many other examples of groups working on stuff that no one would ever use or care about. Mobile IP initiatives,

endless work around standards that nobody cared about, research from the labs that would never be applied or even cited.

“Yikes.

“I had written stuff that people actually used, before. It felt good. I had written a usenet newsreader that was used by hundreds of thousands of people. I was running an online game, as a commercial hobby on the side, which had several hundred paying customers. Sheesh, I thought. *My side projects have more customers than my day job.*

“So I made a simple resolution. I wanted to work on stuff that people would actually use.

“This sounds simple. But if you walk the halls of Sun, AOL, HP, IBM, AOL, Cisco, Siebel, Oracle, any university, many startups, and even Google and Yahoo, you'll find people working on stuff that isn't going to ship. Or that if it does ship, it won't be noticed, or won't move the needle. That's tragic. It's like writing a blog that nobody reads. People make fun of bloggers who are writing ‘only for their mother.’ But what about the legion of programmers writing code paths that will never be traversed?”

It's for precisely this reason that [I've often wondered if writing code is really the most effective way for software developers to spend their time](#). A software developer that doesn't write code—sacrilege, right?

But wait a minute. A smart software developer knows that there's no point in writing code if it's code that nobody will see, code that nobody will use, code that nobody will ultimately benefit from. Why build a permanently vacant house?

A smart software developer realizes that their job is far more than writing code and shipping it; **their job is to build software that people will actually want to use**. That encompasses coding, sure, but it also includes a whole host of holistic, non-coding activities that are critical to the overall success of the software. Things like [documentation](#), [interaction design](#), [cultivating user community](#), all the way up to the [product vision](#) itself. If you get that stuff wrong, it won't matter what kind of code you've written.

If, like Rich Skrenta, you want to work on software that people want to use, realize that it's part of your job to make that software worth using.

OceanofPDF.com

Don't Ask—Observe

James Surowiecki, author of [The Wisdom of Crowds](#), writes about [the paradox of complexity and consumer choice](#) in a recent New Yorker column:

“A recent study by a trio of marketing academics found that when consumers were given a choice of three models, of varying complexity, of a digital device, more than sixty per cent chose the one with the most features. Then, when the subjects were given the chance to customize their product, choosing from twenty-five features, they behaved like kids in a candy store. (Twenty features was the average.) But, when they were asked to use the digital device, so-called "feature fatigue" set in. They became frustrated with the plethora of options they had created, and ended up happier with a simpler product.”

It's impossible to see that you're creating a frankenstein's monster of a product—until you attempt to use it. It's what I call **the all-you-can-eat buffet problem**. There's so much delicious food to choose from at the buffet, and you're so very hungry. Naturally you load up your plate with wild abandon. But after sitting down at the table, you belatedly realize there's no way you could possibly eat all that food.

In all fairness, sometimes people do, in fact, want complexity. The [newly redesigned Google Korea homepage](#) is intentionally complex. Google's Marissa Mayer noted, “it was important where our classic minimalism wasn't working that we adapt.”

Google™

    
웹문서 | 이미지 | 뉴스 | 블로그 | 그룹스 | 더보기 »

Google 검색

I'm Feeling Lucky

전체 웹 한국어 웹 고급검색 | 환경설정 | 언어도구

웹에서 즐기는 채팅과 전화

      
Gmail | 토크 | 캘린더 | 노트 | 툴바 | 데스크톱 | Picasa

채용정보 | 광고 프로그램 | Google 정보 | Google.com in English

©2007 Google

This echoes an [earlier blog post by Donald Norman](#) describing the way South Koreans seek out complexity in luxury goods:

“I recently toured a department store in South Korea. Visiting department stores and the local markets is one of my favorite pastimes whenever I visit a country new to me, the better to get to know the local culture. Foods differ, clothes differ, and in the past, appliances differed: appliances, kitchen utensils, gardening tools, and shop tools.

“I found the traditional ‘white goods’ most interesting: Refrigerators and washing machines. The store obviously had the Korean companies LG and Samsung, but also GE, Braun, and Philips. The Korean products seemed more complex than the non-Korean ones, even though the specifications and prices were essentially identical. ‘Why?’ I asked my two guides, both of

whom were usability professionals. ‘Because Koreans like things to look complex,’ they responded. It is a symbol: it shows their status.’

What's particularly telling in the study Surowiecki cites is the disconnect between what people say they want and what they actually want. You'll find this theme echoed over and over again in usability circles: **what users say they will do, and what they actually do, are often two very different things.** That's why asking users what they want is nearly useless from a usability perspective; **you have to observe what users actually do.** That's what [usability testing](#) is. Instead of asking consumers what features they wanted in a digital camera, the study should have presented them with a few digital camera prototypes and then observed how they were used. Consumers' success or failure interacting with the prototypes tells us more than a thousand surveys, questionnaires, or focus groups ever could. Unfortunately, creating physical prototypes of digital cameras is prohibitively expensive, so it doesn't happen.

Prototyping software, which is [built out of pure thought-stuff](#), is a much easier proposition. [Dare Obasanjo](#) recently pointed out a great paper, [Practical Guide to Controlled Experiments on the Web](#), which makes a strong case for frequent observational A/B usability tests:

“Greg Linden at Amazon created a prototype to show personalized recommendations based on items in a shopping cart. You add an item, recommendations show up; add another item, different recommendations show up. Linden notes that while the prototype looked promising, a marketing senior vice-president was dead set against it, claiming it would distract people from checking out. Greg was forbidden to work on this any further. Nonetheless, Greg ran a controlled experiment, and the feature won by such a wide margin that not having it live was costing Amazon a noticeable chunk of change. With new urgency, shopping cart recommendations launched. Since then, multiple sites have copied cart recommendations.

“The culture of experimentation at Amazon, where data trumps intuition, and a system that made running experiments easy, allowed Amazon to innovate quickly and effectively.”

Why ask users if they'd like recommendations in their shopping carts when you can simply deploy the feature to half your users, then observe what

happens? Web sites are particularly amenable to this kind of observational testing, because it's easy to collect the user action data on the server as a series of HTTP requests. You don't even have to be physically present to "observe" users this way. However, you can perform the same kind of data analysis, with a little care, even if you're deploying a traditional desktop application. Jensen Harris [describes how Microsoft collects user action data in Office 2003](#):

“Suppose you wanted to know what [Office 2000] features people use the most. Well, you start by asking a ‘guru’ who has worked in the product for a long time. ‘Everyone uses AutoText a lot,’ the guru says. The louder the ‘experts’ are, the more their opinions count. Then you move on to the anecdotal evidence: ‘I was home over Christmas, and I saw my mom using Normal View... that's probably what most beginners use.’ And mix in advice from the helpful expert: ‘most people run multi-monitor, I heard that from the guy at Best Buy.’

“SQM, which stands for ‘Service Quality Monitoring’ is our internal name for what became known externally as the [Customer Experience Improvement Program](#). It works like this: Office 2003 users have the opportunity to opt-in to the program. From these people, we collect anonymous, non-traceable data points detailing how the software is used and on what kind of hardware. (Of course, no personally identifiable data is collected whatsoever.)

“As designers, we define data points we're interested in learning about and the software is instrumented to collect that data. All of the incoming data is then aggregated together on a huge server where people like me use it to help drive decisions.

“What kind of data do we collect? We know everything from the frequency of which commands are used to the number of Outlook mail folders you have. We know which keyboard shortcuts you use. We know how much time you spend in the Calendar, and we know if you customize your toolbars. In short, we collect anything we think might be interesting and useful as long as it doesn't compromise a user's privacy.”

This may sound eerily like Big Brother, but the SQM merely extends the same level of reporting enjoyed in every single web application ever created to desktop applications.

The true power of this data is that you can remotely, silently, automatically "observe" what users actually do in your software. Now you can answer questions like [what are the top five most used commands in Microsoft Word 2003?](#) The answer may surprise you. Do you know what the top five most frequently used functions in your application are?

Don't get me wrong. I love users. Some of my best friends are users. But like all of us humans, they're unreliable at best. **In order to move beyond usability guesswork, there's no substitute for observing customers using your product.** Wouldn't it be liberating to be able to make design decisions based on the way your customers actually use your software, rather than the way they tell you they use it? Or the way you think they use it? Whether you're observing users in [low-fi usability tests](#), or collecting user action data so you can observe users virtually, the goal is the same: **don't ask—observe.**

OceanofPDF.com

Are Features the Enemy?

Mark Minasi is [mad as hell](#), and he's not going to take it any more. In his online book [The Software Conspiracy](#), he examines in great detail the paradox I struggled with yesterday—new features are used to sell software, but they're also the primary reason that software [spoils over time](#).

“If a computer magazine publishes a roundup of word processors, the central piece of that article will be the "feature matrix," a table showing what word processing programs have which features. With just a glance, the reader can quickly see which word processors have the richest sets of features, and which have the least features. You can see an imaginary example in the following table:”

| | MyWord 2.1 | BugWord 2.0 | SmartWords 3.0 |
|---|-----------------------|------------------------|---------------------------|
| Can boldface text | X | X | |
| Runs on the Atari 520 | | X | |
| Automatically indents first line of a paragraph | X | | |
| Includes game for practicing touch typing | | X | X |
| Lets you design your own characters | | X | X |
| Generates document tables of contents | X | | |
| Can do rotating 3D bullet points in color | | X | X |
| Can do bulleted lists | X | | |
| Supports Cyrillic symbol set | | X | |
| Includes Malaysian translater | | X | X |

“It looks like BugWord 2.0 is the clear value—there are lots more checkboxes in its column. However, a closer look reveals that it lacks some very basic and useful word processing features, which MyWord 2.1 has. But

the easy-to-interpret visual nature of a feature matrix seems to mean that the magazine's message is: Features are good, and the more the better. As Internet Week senior executive editor Wayne Rash, a veteran of the computer press, says, "Look at something like PC Magazine, you'll see this huge comparison chart. Every conceivable feature any product could ever do shows up, and if a package has that particular feature, then there's a little black dot next to that product. What companies want is to have all the little black dots filled in because it makes their software look better."

Mark maintains that software companies give bugs in their existing software a low priority, while developing new features for the next version is considered critically important. As a result, quality suffers. He trots out this Bill Gates quote as a prime example:

“There are no significant bugs in our released software that any significant number of users want fixed... The reason we come up with new versions is not to fix bugs. It's absolutely not. It's the stupidest reason to buy a new version I ever heard... And so, in no sense, is stability a reason to move to a new version. It's never a reason.”

It's hard to argue with the logic. Customers will pay for new features. But customers will never pay companies to fix bugs in their software. Unscrupulous software companies can exploit this by fixing bugs in the next version, which just so happens to be jam packed full of exciting new features that will induce customers to upgrade.

Unlike Mark, I'm not so worried about bugs. All software has bugs, and if you accrue enough of them, your users will eventually revolt. Yes, the financial incentives for fixing bugs are weak, but the market seems to work properly when faced with buggy software.

A much deeper concern, for me, is **the subtle, creeping feature-itis that destroys my favorite software**. It's the worst kind of affliction—a degenerative disease that sets in over time. As I've regrettably discovered in many, many years of using software, adding more features rarely results in better software. The commercial software market, insofar as it forces vendors to engage in bullet point product feature one-upsmanship, could be actively harming the very users it is trying to satisfy.

And the worst part, the absolute worst part, is that customers are complicit in the disease, too. Customers ask for those new features. And customers will use the dreaded "feature matrix" as a basis for comparing what applications they'll buy. Little do they know that they're slowly killing the very software that they love.

Today, as I was starting up WinAmp, I was blasted by this upgrade dialog:



Do I care about any of these new features? No, not really. Album art sounds interesting, but the rest are completely useless to me. I don't have to upgrade, of course, and there's nothing forcing me to upgrade. Yet. My concern here isn't for myself, however. It's for WinAmp. For every new all-singing, all-dancing feature, WinAmp becomes progressively slower, even larger, and more complicated. Add enough well-intentioned "features," and eventually WinAmp will destroy itself.

Sometimes, I wonder if the current commercial software model is doomed. The neverending feature treadmill it puts us on almost always results in extinction. Either the application eventually becomes so bloated and ineffective that smaller, nimbler competitors replace it, or the application slowly implodes under its own weight. In either case, nothing is truly fixed; the cycle starts anew. Something always has to give in the current model. Precious few commercial software packages are still around after 10 years, and most of the ones that are feel like dinosaurs.

Perhaps we should **stop blindly measuring software as a bundle of features**, as some kind of endless, digital all-you-can eat buffet. Instead, we could measure software by results—how productive or effective it makes us at whatever task we're doing. Of course, measuring productivity and results is hard, whereas counting bullets on a giant feature matrix is brainlessly easy. Maybe that's exactly the kind of cop-out that got us where we are today.

OceanofPDF.com

The Organism Will Do Whatever It Damn Well Pleases

In the go-go world of software development, we're so consumed with [learning new things](#), so [fascinated with the procession of shiny new objects](#) that I think we sometimes lose sight of our history. I don't mean the big era-defining successes. Everyone knows those stories. I'm talking about the things we've tried before that... didn't quite work out. The failures. The also-rans. The noble experiments. The crazy plans.

I'm all for [reinventing the wheel](#), because it's one of the best ways to learn. But you shouldn't even think about reinventing a damn thing **until you've exhaustively researched every single last wheel, old or new, working or broken, that you can lay your hands on.** Do your homework.

That's why I love unearthing stories like [The Lessons of Lucasfilm's Habitat](#). It's basically World of Warcraft... in 1985.

“[Habitat](#) is ‘a multi-participant online virtual environment,’ a cyberspace.”



“Each participant (‘player’) uses a home computer (Commodore 64) as an intelligent, interactive client, communicating via modem and telephone over a commercial packet-switching network to a centralized, mainframe host system. The client software provides the user interface, generating a real-time animated display of what is going on and translating input from the player into messages to the host. The host maintains the system's world model enforcing the rules and keeping each player's client informed about the constantly changing state of the universe.”

This was the dark ages of home computing. In 1985, that 64k of memory in a Commodore 64 was a lot. The entirety of Turbo Pascal 3.02 for DOS, released a year later in 1986, [was barely 40k](#).

The very concept of a multiplayer virtual world of any kind—something we take for granted today, since every modern website is essentially a multiplayer game now—was incredibly exotic. Look at the painstaking explanation Lucasfilm had to produce to even get folks to understand what the heck Habitat was, and how it worked:

The technical information in [The Lessons of Lucasfilm's Habitat](#) is incredibly dated, as you'd expect, and barely useful even as trivia. But the sociological lessons of Habitat cut to the bone. They're as fresh today as they were in 1985. Computers have radically changed in the intervening 27 years, whereas people's behavior hasn't. At all. This particular passage hit home:

“Again and again we found that activities based on often unconscious assumptions about player behavior had completely unexpected outcomes (when they were not simply outright failures). It was clear that we were not in control. The more people we involved in something, the less in control we were. We could influence things, we could set up interesting situations, we could provide opportunities for things to happen, but we could not predict nor dictate the outcome. Social engineering is, at best, an inexact science, even in proto-cyberspaces. Or, as some wag once said, ‘in the most carefully constructed experiment under the most carefully controlled conditions, the organism will do whatever it damn well pleases.’”

Even more specifically:

“Propelled by these experiences, we shifted into a style of operations in which we let the players themselves drive the direction of the design. This

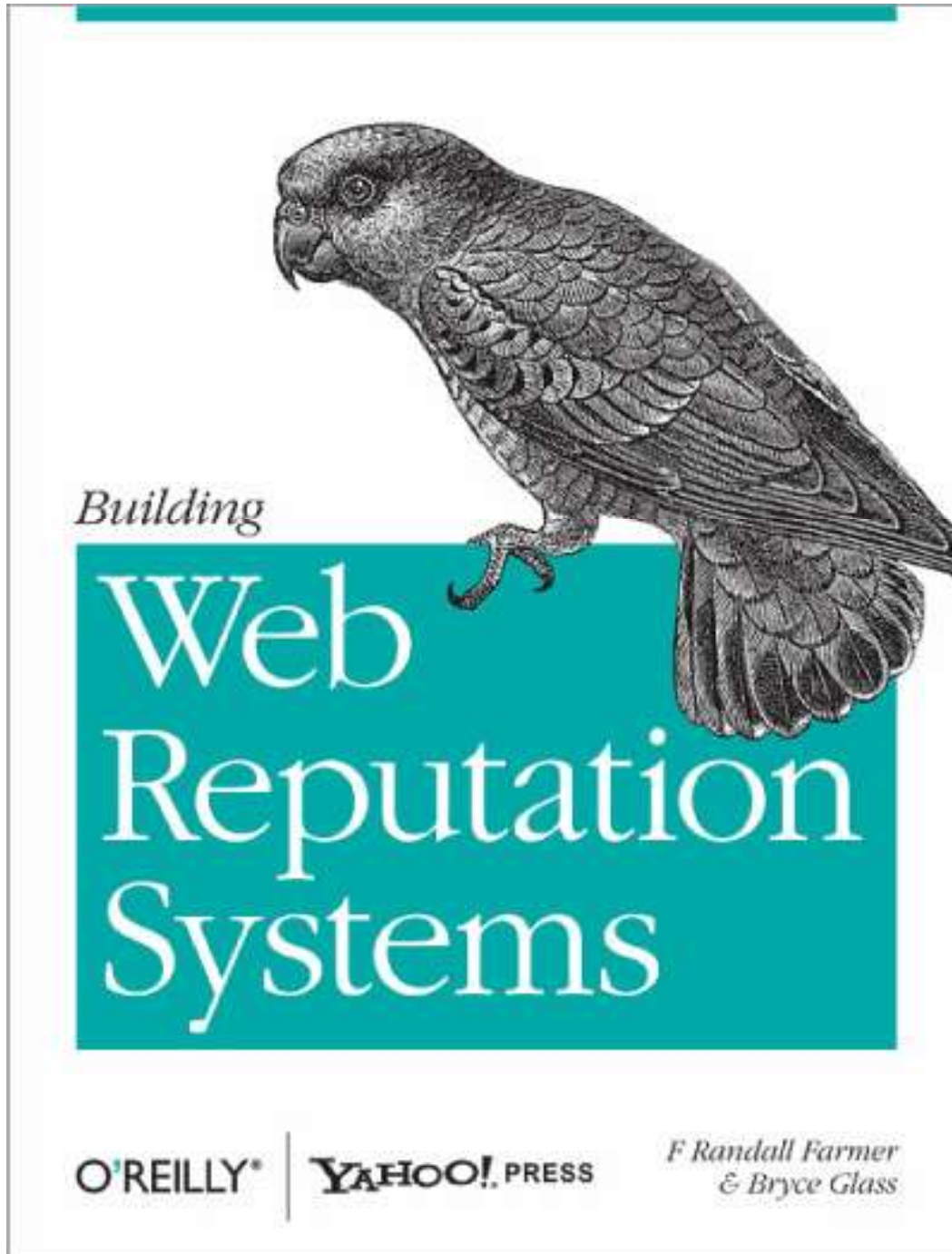
proved far more effective. Instead of trying to push the community in the direction we thought it should go, an exercise rather like herding mice, we tried to observe what people were doing and aid them in it. We became facilitators as much as designers and implementors. This often meant adding new features and new regions to the system at a frantic pace, but almost all of what we added was used and appreciated, since it was well matched to people's needs and desires. As the experts on how the system worked, we could often suggest new activities for people to try or ways of doing things that people might not have thought of. In this way we were able to have considerable influence on the system's development in spite of the fact that we didn't really hold the steering wheel—more influence, in fact, than we had had when we were operating under the delusion that we controlled everything.”

That's exactly what I was trying to say in [Listen to Your Community, But Don't Let Them Tell You What to Do](#). Unfortunately, because I hadn't read this essay until a few months ago, I figured this important lesson out 25 years later than Randy Farmer and Chip Morningstar. So many Stack Overflow features were the direct result of observing what the community was doing, then attempting to aid them in it:

- We noticed early in the Stack Overflow beta that users desperately wanted to reply to each other, and were cluttering up the system with "answers" that were, well, not answers to the question. Rather than chastise them for doing it wrong—stupid users!—we added the commenting system to give them a method of annotating answers and questions for clarifications, updates, and improvements.
- I didn't think it was necessary to have a place to discuss Stack Overflow. And I was... kind of a jerk about it. The community was on the verge of creating a phpBB forum instance to discuss Stack Overflow. Faced with a nuclear ultimatum, I relented, and you know what? [They were right](#). And I was wrong.
- The community came up with an interesting convention for [handling duplicate questions](#), by manually editing a blockquote into the top of the question with a link to the authoritative question that it was a duplicate of. This little user editing convention eventually became the template for the official implementation.

I could go on and on, but I won't bore you. I'd say for every 3 features we introduced on Stack Overflow, at least two of them came more or less directly from observing the community, then trying to run alongside them, building tools that **helped them do what they wanted to do with less fuss and effort**. That was my job for the last four years. And I loved it, until I had to [stop loving it](#).

[Randy Farmer](#), one of the primary designers of Habitat at Lucasfilm, went on to work on a bunch of things that you may recognize: with [Douglas Crockford](#) on JSON, The Sims Online, Second Life, Yahoo 360°, Yahoo Answers, Answers.com, and so forth. He eventually condensed some of his experience into a book, [Building Web Reputation Systems](#), which I bought in April 2011 as a Kindle edition. I didn't know who Mr. Farmer was at this time. I just saw a new O'Reilly book on an area of interest, and I thought I'd check it out.



As the co-founder of Stack Overflow, I know a thing or two about web reputation systems! Out of curiosity, I looked up the author on my own site. And [I found him](#), with a tiny reputation. So I sent this friendly jibe on Twitter:

 **Jeff Atwood** @codinghorror 11 Apr 11
pff, look at [@frandallfarmer](#) 's tiny rep! LOOK AT IT!
<http://goo.gl/Gjvrn> <http://buildingreputation.com/>
Expand

 **F. Randall Farmer** @frandallfarmer 11 Apr 11
[@codinghorror](#) lol! Reputation context is everything! Surprise!
rpg.stackexchange.com/users/810/f-ra...
Hide conversation Reply Retweet Favorite

3 RETWEETS 6 FAVORITES 

6:12 PM - 11 Apr 11 - Details

But the last laugh was on Randy, as it should be, because I didn't realize he had over [6,000 reputation](#) on rpg.stackexchange.com. Turns out, Randy Farmer was already an avid Stack Exchange user. And, as you might guess given his background, a rather expert Stack Exchange user at that. The Stack Exchange ruleset is complex, strict, and requires discipline to understand. Kind of like.. maybe a certain role playing game, if you will.

ADVANCED DUNGEONS & DRAGONS®



SPECIAL REFERENCE WORK

PLAYERS HANDBOOK

A COMPILED VOLUME OF INFORMATION FOR PLAYERS OF
ADVANCED DUNGEONS & DRAGONS, INCLUDING: CHARACTER RACES,
CLASSES, AND LEVEL ABILITIES; SPELL TABLES AND DESCRIPTIONS;
EQUIPMENT COSTS; WEAPONS DATA; AND INFORMATION ON ADVENTURING.

by Gary Gygax

© 1978 — TSR Games

All rights reserved

Illustrations by David C. Sutherland III

D. A. Trampier

Cover by D. A. Trampier

Advanced Dungeons & Dragons is a registered trademark of TSR, Inc. All rights reserved. © 1978 TSR, Inc. All rights reserved.

This book is published under the copyright laws of the United States of America. Any reproduction in other countries without the written or printed consent of the publisher is prohibited without the express written permission of TSR, Inc.

TSR, Inc. is a U.S.A. corporation.
© 1978 TSR, Inc.

Randy is the sort of dad who [had his first edition Dungeons & Dragons books bound into a single leather tome and handed it down to his son as a family heirloom](#). How awesome is that?

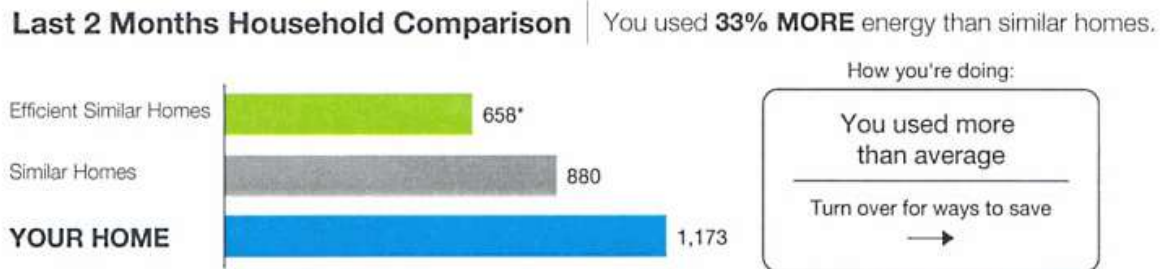
If we've learned anything in the last 25 years since Habitat, it is that **people are the source of, and solution to, all the problems you'll run into when building social software.** Are you looking to (dungeon) master the art of guiding and nudging your online community through their collective adventure, without violating the continuity of your own little universe? If so, you could do a whole heck of lot worse than reading [Building Web Reputation Systems](#) and following [@FRandallFarmer](#) on Twitter.

[OceanofPDF.com](#)

For a Bit of Colored Ribbon

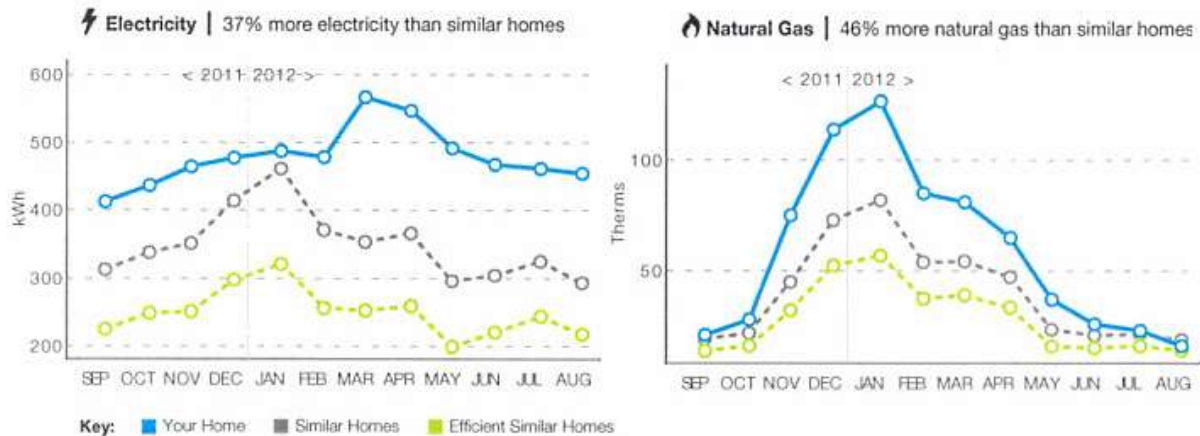
For the last year or so, I've been getting these two page energy assessment reports in the mail from Pacific Gas & Electric, our California utility company, comparing our household's energy use to those of the houses around us.

Here's the relevant excerpts from the latest report; click through for a full-page view of each page.



* This energy index combines electricity (kWh) and natural gas (therms) into a single measurement.

- **Similar Homes:** Approximately 100 occupied nearby homes that are similar in size to yours (avg 1,196 sq ft) and have gas heat
- **Efficient Similar Homes:** The most efficient 20 percent of similar homes



These poor results are particularly galling because I go far out of my way to Energy Star all the things, I use LED light bulbs just about everywhere, we set our thermostat appropriately, and we're still getting crushed. I have no particular reason to care about this stupid energy assessment report showing

our household using 33 percent more energy than similar homes in our neighborhood. **And yet... I must win this contest.** I can't let it go.

- Installed a [Nest 2.0 learning thermostat](#).
- I made sure every last bulb in our house that gets any significant use is LED. Fortunately there are some pretty decent [\\$16 LED bulbs on Amazon now](#) offering serviceable 60 watt equivalents at 9 watt, without too many early adopter LED quirks (color, dimming, size, weight, etc).
- I even put [appliance LED bulbs](#) in our refrigerator and freezer.
- Switched to a [low-flow shower head](#).
- Upgraded to a high efficiency tankless water heater, the [Noritz NCC1991-SV](#).
- Nearly killed myself trying to source LED candelabra bulbs for the fixture in our dining room which has 18 of the damn things, and is used quite a bit now with [the twins](#) in the house. Turns out, 18 times any number ... is still kind of a large number. In cash.

(Most of this has not helped much on the report. The jury is still out on the Nest thermostat and the candelabra LED bulbs, as I haven't had them long enough to judge. I'm gonna [defeat this thing, man!](#))

I'm ashamed to admit that it's only recently I realized that this technique—showing a set of metrics alongside your peers—is exactly the same thing we built at Stack Overflow and [Stack Exchange](#). Notice any resemblance on the user profile page here?

2,428 Answers

votes activity newest

- 173 Should I use != or <> for not equal in TSQL?
- 170 "INSERT IGNORE" vs "INSERT ... ON DUPLICATE KEY UPD..."
- 165 Fetch the row which has the Max value for a column
- 164 What's the difference between identifying and non-identifying rel...
- 133 How do the Proxy, Decorator, Adaptor, and Bridge Patterns diff...

[view more](#)

12 Questions

votes activity newest

- 85 Why is debugging better in an IDE?
- 67 Tools for Generating Mock Data?
- 18 Emulate MySQL LIMIT clause in Microsoft SQL Server 2000
- 17 Dump Django site to static HTML? [closed]
- 10 Pythonic equivalent of unshift or redo?

[view more](#)

9 Accounts

| | | |
|---|-------------|----------------|
|  Stack Overflow | 120,280 rep | 19 • 116 • 296 |
|  Programmers | 1,261 rep | • 6 • 17 |
|  Meta Stack Overflow | 661 rep | • 6 • 8 |
|  Database Administrators | 646 rep | • 1 • 4 • 8 |
|  Home Improvement | 271 rep | • 2 • 4 |

120,280 Reputation

top 0.05% overall



- +10 Zend DB Framework examine query for an update
- +10 make an ID in a mysql table auto_increment (after the fact)
- +10 What is the most efficient/elegant way to parse a flat table into a L...
- +10 What's the difference between identifying and non-identifying rel...

[view more](#)

1,036 Tags

| | |
|--|--|
| 4k <code>mysql</code> × 1196 | 672 <code>zend-framework</code> × 132 |
| 3k <code>sql</code> × 753 | 640 <code>greatest-n-per-group</code> × 83 |
| 2k <code>php</code> × 626 | 507 <code>query</code> × 123 |
| 1k <code>database</code> × 303 | 454 <code>sql-server</code> × 132 |
| 878 <code>database-design</code> × 152 | 354 <code>join</code> × 80 |

[view more](#)

93 Badges

recent class name

| | |
|-------------------------|--------------------|
| • Popular Question × 10 | • Yearling × 4 |
| • Great Answer × 8 | • table |
| • Nice Question × 6 | • Guru × 17 |
| • Enlightened × 40 | • Good Answer × 53 |
| • Nice Answer × 216 | • mysql |

You've tricked me into becoming obsessed with understanding and reducing my household energy consumption. Something that not only benefits me, but also benefits the greater community and, more broadly, benefits the entire world. You've beaten me at my own game. Well played, Pacific Gas and Electric. Well played.



David Copeland
@davetron5000

@codinghorror @simucal used to work at the company that sends those. The reports reduce consumption continuously over several years. It works

← Reply ↻ Retweet ★ Favorite

1:25 PM - 25 Sep 12 📍 from Washington, DC · Embed this Tweet

This peer motivation stuff, [call it gamification if you must](#), really works. That's why we do it. But these systems are like firearms: so powerful they're kind of dangerous if you don't know what you're doing. If you don't think deeply about what you're incentivizing, why you're incentivizing it, and the full ramifications of all emergent behaviors in your system, you may end up with... something darker. [A lot darker](#).

“The key lesson for me is that our members became very thoroughly obsessed with those numbers. Even though points on Consumating were redeemable for absolutely nothing, not even a gold star, our members had an unquenchable desire for them. What we saw as our membership scrabbled over valueless points was that there didn't actually need to be any sort of material reward other than the points themselves. We didn't need to allow them to trade the points in for benefits, virtual or otherwise. It was enough of a reward for most people just to see their points wobble upwards. If only we had been able to channel that obsession towards something with actual value!”

Since [I left Stack Exchange](#), I've had a difficult time explaining what exactly it is I do, if anything, to people. I finally settled on this: what I do, what I'm best at, what I love to do more than anything else in the world, is **design massively multiplayer games for people who like to type paragraphs to each other**. I channel their obsessions—and mine—into something positive,

something that they can learn from, something that creates wonderful reusable artifacts for the whole world. And that's what I still hope to do, because I have an endless well of obsession left.

Just ask PG and E.

OceanofPDF.com

Building Social Software for the Anti-Social

In November, I delivered the keynote presentation at [Øredev](#) 2011. It was the second and probably final presentation in the series I call **Building Social Software for the Anti-Social**.

I've spent almost four years thinking about the Q&A format, and these two presentations are the culmination of that line of thought. In them I present ten "scary ideas," ideas which are counterintuitive for most folks. These are the building blocks we used to construct Stack Overflow, and by extension, Server Fault, Super User, and the rest of the [Stack Exchange network](#).

1. Radically lower the bar for participation.
2. Trusting (some of) your users.
3. Life is the world's biggest MMORPG.
4. Bad stuff happens.
5. Love trumps money.
6. Rules can be fun and social.
7. All modern website design is game design.
8. Thoughtful game design creates sustainable communities.
9. The community isn't always right.
10. Some moderation required.

It's not the same experience as attending the actual live presentation, of course, but you can certainly get the gist of it by viewing the slides for these two presentations online:



Social software for the anti-social
(programmers)

Building Social Software for the
Anti-Social

Part II



Jeff Atwood
stackexchange.com

The Øredev organizers hired [ImageThink](#) to draw each presentation on a whiteboard live on stage as it was presented. I was skeptical that this would

work, but the whiteboard visualizations came out great for all the presentations. Here's the two whiteboard drawings ImageThink created during my presentation. (Yes, they had two artists on stage "live whiteboarding," one on the left side, and one on the right side.)





It's not a bad approximation of what was covered. If you're curious about live whiteboard visualizations, ImageThink posted [a great set of links on their blog](#) that I highly recommend.

After four years, we've mostly figured out what works and what doesn't work for our particular brand of low noise, high signal Q and A at Stack Exchange. But the title **Social Software for the Anti-Social** is only partially tongue in cheek. If you want to learn anything online, you have to design your software to refocus and redirect people's natural social group impulses, and that's what these presentations attempt to explain. I hope you enjoy them!

Update: Part II is now available as a full talk, with audio and video courtesy of Oredev. [Watch it now!](#)

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

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

OceanofPDF.com

VI.

Causes We Should Care About

OceanofPDF.com

Preserving the Internet... And Everything Else

In [Preserving Our Digital Pre-History](#) I nominated [Jason Scott](#) to be our generation's **digital historian in residence**. It looks like a few people must have agreed with me, because in March 2011, he [officially became an archivist](#) at the Internet Archive.



Jason recently invited me to visit the [Internet Archive](#) office in nearby San Francisco. The building alone is amazing; when you imagine the place where they store the entire freaking Internet, this enormous [former Christian Science church](#) seems... well, about right.

It's got a built in evangelical aura of mission, with new and old computer equipment strewn like religious totems throughout.



Doesn't it look a bit like the place where we worship servers, with Jason Scott presiding over the invisible, omnipresent online flock? It's all that and so much more. Maybe the religious context is appropriate, because I always thought the Internet Archive's mission—to **create a permanent copy of every Internet page ever created, as it existed at the time**—was audacious bordering on impossible. You'd need to be a true believer to even consider the possibility.

The Internet Archive is about the only ally we have in the fight against pernicious and pervasive [linkrot](#) all over the Internet. When I go back and [review old Coding Horror blog entries](#) I wrote in 2007, it's astonishing just how many of the links in those posts are now, after five years, gone. I've lost count of all the times I've used the [Wayback Machine](#) to retrieve historical Internet pages I once linked to that are now permanently offline—pages that would have otherwise been lost forever.

“The Internet Archive is a service so essential that its founding is bound to be looked back on with the fondness and respect that people now have for the public libraries seeded by Andrew Carnegie a century ago... Digitized information, especially on the Internet, has such rapid turnover these days that total loss is the norm. Civilization is developing severe amnesia as a result; indeed it may have become too amnesiac already to notice the problem properly. The Internet Archive is the beginning of a cure—the beginning of complete, detailed, accessible, searchable memory for society, and not just scholars this time, but everyone.”—[Stewart Brand](#)

Without the Internet Archive, the Internet would have no memory. As [the world's foremost expert on backups](#) I cannot emphasize enough how significant the Internet Archive is to the world, to any average citizen of the Internet who needs to source an old hyperlink. Yes, maybe it is just the world's largest and most open hard drive, but nobody else is doing this important work that I know of.

Let's Archive Atoms, Too

While what I wrote above is in no way untrue, it is only a small part of the Internet Archive's mission today. Where I always thought of the Internet Archive as, well, an archive of the bits on the Internet, they have long since broadened the scope of their efforts to include stuff made of filthy, dirty, nasty atoms. Stuff that was never on the Internet in the first place.

The Internet Archive isn't merely archiving the Internet any more, **they are attempting to archive everything.**

- [Open Library](#)—one web page for every book ever published.
- [Live Music Archive](#)—every live concert ever recorded, free for non-commercial use.
- [Physical Archive](#)—one physical copy of every book ever published.
- [Moving Image Archive](#)—every free movie, film, or video ever recorded.
- [Text Archive](#)—with 1.6 million scanned books already online, and a thousand more [scanned in every day](#).

All of this, in addition to boring mundane stuff like taking snapshots of the entire Internet every so often. That's going to take, uh... a lot of hard drives. I snapped a picture of a giant pile of 3 TB drives waiting to be installed in one of the storage rooms.



The Internet Archive is a big organization now, with 30 employees in the main San Francisco office you're seeing above, and 200 staff all over the world. With a mission of such overwhelming scope and scale, they're going to need all the help they can get.

The Internet Archive Needs You

The Internet Archive is a non-profit organization, so you could certainly [donate money](#). If your company does charitable donations and cares at all about the Internet, or free online access to human knowledge, I'd strongly

encourage them to donate to the Internet Archive as well. I made sure that Stack Exchange [donated](#) every year.

But more than money, what the Internet Archive needs these days is ... your stuff. I'll let Jason explain exactly what he's looking for:

“I'm trying to acquire as much in the way of obscure video, obscure magazines, unusual pamphlets and printed items of a computer nature or even of things like sci-fi, zines—anything that wouldn't normally find itself inside most libraries. Hence [my computer magazines collection](#)—tens of thousands of issues in there. I'd love to get my hands on more.

Also as mentioned, I love, love, *love* shareware CDs. Those are the most bang for the buck with regards to data and history that I want to get my hands on.”

Being the obsessive conscientious geeks that I know you are, I bet you have a collection of geeky stuff exactly like that somewhere in your home. If so, the best way you can help is to send it in as a contribution! **Email jscott@archive.org** about what you have, and if you're worried about rejection, don't be:

“There's seriously nothing we don't want. I don't question. I take it in, I put it in items. I am voracious. Omnivorous. I don't say no.”

The Internet Archive has an impossible mission on an immense scale. It is an unprecedented kind of open source archiving, not driven by Google or Microsoft or some other commercial entity with ulterior motives, but a non-profit organization motivated by nothing more than the obvious common good of building a massive digital [Library of Alexandria](#) to preserve our history for future generations. Let's do our part to help [support the important work of the Internet Archive](#) in whatever way we can.

OceanofPDF.com

The Importance of Net Neutrality

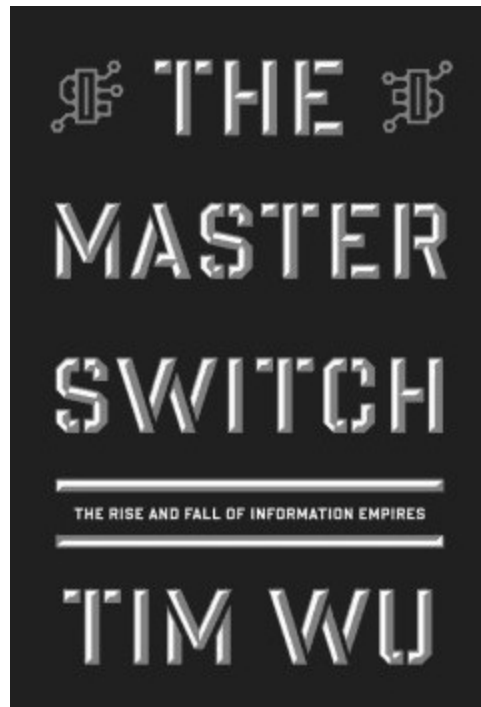
Although I remain [a huge admirer of Lawrence Lessig](#), I am ashamed to admit that **I never fully understood the importance of net neutrality until last week**. Mr. Lessig [described network neutrality in these urgent terms in 2006](#):

“At the center of the debate is the most important public policy you've probably never heard of: ‘network neutrality.’ Net neutrality means simply that all like Internet content must be treated alike and move at the same speed over the network. The owners of the Internet's wires cannot discriminate. This is the simple but brilliant ‘end-to-end’ design of the Internet that has made it such a powerful force for economic and social good: All of the intelligence and control is held by producers and users, not the networks that connect them.”

Fortunately, the good guys are winning. Recent legal challenges to network neutrality have been defeated, [at least under US law](#). I remember hearing about these legal decisions at the time, but I glossed over them because I thought they were fundamentally about file sharing and BitTorrent. Not to sound dismissive, but someone's legal right to download a complete video archive of Firefly wasn't exactly keeping me up at night.

But network neutrality is about far more than file sharing bandwidth. To understand what's at stake, study the sordid history of the world's communication networks—starting with the telegraph, radio, telephone, television, and onward. Without historical context, it's impossible to appreciate how scarily easy it is for [common carriage](#) to get subverted and undermined by corporations and government in subtle (and sometimes not so subtle) ways, with terrible long-term consequences for society.

That's the genius of Tim Wu's book “[The Master Switch: The Rise and Fall of Information Empires](#).”



One of the most fascinating stories in the book is that of [Harry Tuttle and AT&T](#).

“Harry Tuttle was, for most of his life, president of the Hush-a-Phone Corporation, manufacturer of the telephone silencer. Apart from Tuttle, Hush-a-Phone employed his secretary. The two of them worked alone out of a small office near Union Square in New York City. Hush-a-Phone's signature product was shaped like a scoop, and it fit around the speaking end of a receiver, so that no one could hear what the user was saying on the telephone. The company motto emblazoned on its letterhead stated the promise succinctly: ‘Makes your phone private as a booth.’

“If the Hush-a-Phone never became a household necessity, Tuttle did a decent business, and by 1950 he would claim to have sold 125,000 units. But one day late in the 1940s, Henry Tuttle received alarming news. AT&T had launched a crackdown on the Hush-a-Phone and similar products, like the Jordaphone, a creaky precursor of the modern speakerphone, whose manufacturer had likewise been put on notice. Bell repairmen began warning customers that Hush-a-Phone use was a violation of a federal tariff and that, failing to cease and desist, they risked termination of their telephone service.

“Was AT&T merely blowing smoke? Not at all: the company was referring to a special rule that was part of their covenant with the federal government. It stated: *‘No equipment, apparatus, circuit or device not furnished by the telephone company shall be attached to or connected with the facilities furnished by the telephone company, whether physically, by induction, or otherwise.’*”

“Tuttle hired an attorney, who petitioned the FCC for a modification of the rule and an injunction against AT&T's threats. In 1950 the FCC decided to hold a trial (officially a ‘public hearing’) in Washington, D.C., to consider whether AT&T, the nation's regulated monopolist, could punish its customers for placing a plastic cup over their telephone mouthpiece.

“The story of the Hush-a-Phone and its struggle with AT&T, for all its absurdist undertones, offers a window on the mindset of the monopoly at its height, as well as a picture of the challenges facing even the least innovative innovator at that moment.”

Absurdist, indeed—Harry Tuttle is also not-so-coincidentally the name of a character in the movie [Brazil](#), one who attempts to work as a renegade, outside oppressive centralized government systems. Often at great peril to his own life and, well, that of anyone who happens to be nearby, too.



But the story of Harry Tuttle isn't just a cautionary tale about the dangers of large communication monopolies. Guess who was on Harry Tuttle's side in his [sadly doomed legal effort against the enormously powerful Bell monopoly](#)? No less than an acoustics professor by the name of **Leo Beranek**, and an expert witness by the name of **J.C.R. Licklider**.

If you don't recognize those names, you should. [J.C.R. Licklider](#) went on to propose and design ARPANET, and [Leo Beranek](#) became one of the B's in [Bolt, Beranek and Newman](#), who helped build ARPANET. In other words, these gentlemen went on from battling the Bell monopoly in court in the 1950s to designing a system in 1968 that would ultimately defeat it: the internet.

The internet is radically unlike all the telecommunications networks that have preceded it. It's the first national and global communication network designed from the outset to resist mechanisms for centralized control and monopoly. But resistance is not necessarily enough; [The Master Switch](#) makes a compelling case that, historically speaking, all communication networks **start out open and then rapidly swing closed as they are increasingly commercialized**.

“Just as our addiction to the benefits of the internal combustion engine led us to such demand for fossil fuels as we could no longer support, so, too, has our dependence on our mobile smart phones, touchpads, laptops, and other devices delivered us to a moment when our demand for bandwidth—the new black gold—is insatiable. Let us, then, not fail to protect ourselves from the will of those who might seek domination of those resources we cannot do without. If we do not take this moment to secure our sovereignty over the choices that our information age has allowed us to enjoy, we cannot reasonably blame its loss on those who are free to enrich themselves by taking it from us in a manner history has foretold.”

It's up to us to be vigilant in protecting the concepts of common carriage and network neutrality on the internet. Even devices that you may love, like an iPad, Kindle, or Xbox, can easily be turned against you—if you let them.

Youtube vs. Fair Use

In [YouTube: The Big Copyright Lie](#), I described my love-hate relationship with YouTube, at least as it existed in way back in the dark ages of 2007.

“Now think back through all the videos you've watched on YouTube. How many of them contained any original content?”

“It's perhaps the ultimate case of cognitive dissonance: by YouTube's own rules [which prohibit copyrighted content], YouTube cannot exist. And yet it does.

“How do we reconcile YouTube's official hard-line position on copyright with the reality that 90 percent of the content on their site is clearly copyrighted and clearly used without permission? It seems YouTube has an awfully convenient "don't ask, don't tell" policy—they make no effort to verify that the uploaded content is either original content or fair use. The copyrighted content stays up until the copyright owner complains. Then, and only then, is it removed.”

Today's lesson, then, is **be careful what you ask for**.

At the time, I just assumed that YouTube would never be able to resolve this problem through technology. The idea that you could somehow fingerprint every user-created uploaded video against every piece of copyrighted video ever created was so laughable to me that I wrote it off as impossible.

I uploaded a small clip from the movie “[Better Off Dead](#)” to YouTube, in order to use it in a blog entry. This is quintessential **fair use**: a tiny excerpt of the movie, presented in the context of a larger blog entry. So far, so good.

But then I uploaded a small clip from a different movie that I'm planning to use in another, future blog entry. Within an hour of uploading it, I received this email:

Dear {username},

Your video, {title}, may have content that is owned or licensed by {company}.

No action is required on your part; however, if you are interested in learning how this affects your video, please visit [the Content ID Matches section of your account](#) for more information.

Sincerely,

-The YouTube Team

This 90 second clip is from a recent movie. Not a hugely popular movie, mind you, but a movie you've probably heard of. This email both fascinated and horrified me. **How did they match a random, weirdly cropped (thanks, Windows Movie Maker) clip from the middle of a non-blockbuster movie** within an hour of me uploading it? This had to be some kind of automated process that checks uploaded user content against every piece of copyrighted content ever created (or the top n subset thereof), exactly the kind that I thought was impossible.

Uh oh.

I began to do some research. I quickly found [Fun with YouTube's Audio Content ID System](#), which doesn't cover video, but it's definitely related:

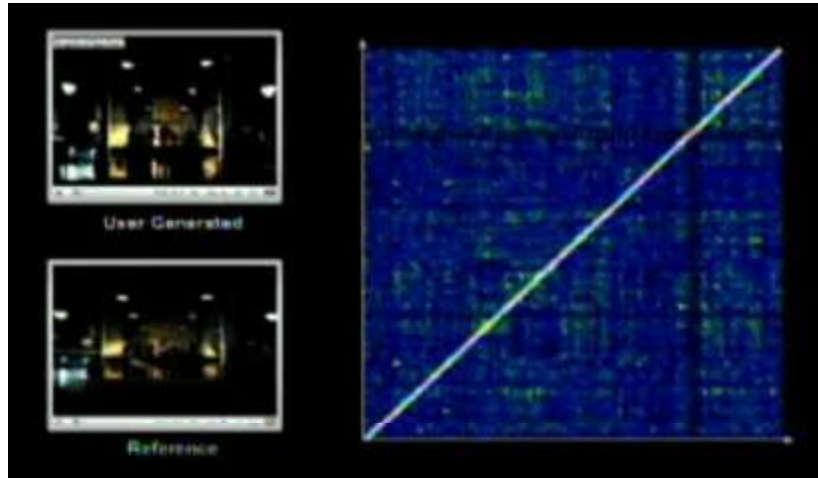
I was caught by surprise one day when I received an automated email from YouTube informing me that my video had a music rights issue and it was removed from the site. I didn't really care.

Then a car commercial parody I made (arguably one of my better videos) was taken down because I used an unlicensed song. That pissed me off. I couldn't easily go back and re-edit the video to remove the song, as the source media had long since been archived in a shoebox somewhere. And I couldn't simply re-upload the video, as it got identified and taken down every time. I needed to find a way to outsmart the fingerprinter. I was angry and I had a lot of free time. Not a good combination.

I racked my brain trying to think of every possible audio manipulation that might get by the fingerprinter. I came up with an almost-scientific method for testing each modification, and I got to work.

Further research led me to this brief TED talk, [How YouTube Thinks About Copyright](#).

“We compare each upload against all the reference files in our database. This heat map is going to show you how the brain of this system works.”



“Here we can see the reference file being compared to the user generated content. The system compares every moment of one to the other to see if there's a match. This means we can identify a match even if the copy uses just a portion of the original file, plays it in slow motion, and has degraded audio or video.

“The scale and speed of this system is truly breathtaking—we're not just talking about a few videos, we're talking about over 100 years of video every day between new uploads and the legacy scans we regularly do across all of the content on the site. And when we compare those 100 years of video, we're comparing it against millions of reference files in our database. It'd be like 36,000 people staring at 36,000 monitors each and every day without as much as a coffee break.”

I have to admit that I'm astounded by the scope, scale, and sheer effectiveness of YouTube's new copyright detection system that I thought was impossible! Seriously, [watch the TED talk](#). It's not long. The more I researched [YouTube's video identification tool](#), the more I realized that **resistance is futile**. It's so good that the only way to defeat it is by degrading your audio and video so much that you have effectively ruined it. And when it comes to copyright violations, if you can achieve mutually assured destruction, then you have won. Absolutely and unconditionally.

This is an outcome so incredible I am still having trouble believing it. But I have the automatically blocked uploads to prove it.

Now, **I am in no way proposing that copyright is something we should be trying to defeat or work around.** I suppose I was just used to the [laissez faire](#) status quo on YouTube, and the idea of a video copyright detection system this effective was completely beyond the pale. My hat is off to the engineers at Google who came up with this system. They aren't the bad guys here; they offer some rather sane alternatives when copyright matches are found:

“If Content ID identifies a match between a user upload and material in the reference library, it applies the usage policy designated by the content owner. The usage policy tells the system what to do with the video. Matches can be to only the audio portion of an upload, the video portion only, or both.

“There are three usage policies—Block, Track or Monetize. If a rights owner specifies a Block policy, the video will not be viewable on YouTube. If the rights owner specifies a Track policy, the video will continue to be made available on YouTube and the rights owner will receive information about the video, such as how many views it receives. For a Monetize policy, the video will continue to be available on YouTube and ads will appear in conjunction with the video. The policies can be region-specific, so a content owner can allow a particular piece of material in one country and block the material in another.”

The particular content provider whose copyright I matched chose the draconian block policy. That's certainly not Google's fault, but I guess you could say I'm Feeling Unlucky.

Although the 90 second clip I uploaded is clearly copyrighted content—I would never dispute that—**my intent is not to facilitate illegal use, but to "quote" the movie scene as part of a larger blog entry.** YouTube does provide recourse for uploaders; they make it easy to file a dispute once the content is flagged as copyrighted. So I dutifully filled out the dispute form, indicating that I felt I had a reasonable claim of fair use.

Dispute Claim (Step 1 of 2)

All fields required.

User Name: codinghorror1

Video ID: F5H18UzTycQ

Select the reason for your dispute.

- 1. This video does not feature the third-party copyrighted material at issue. My video was misidentified as containing this material.
- 2. This video uses copyrighted material in a manner that does not require approval of the copyright holder. It is a fair use under copyright law.

Please explain briefly: 90 sec excerpt to be used in editorial blog

- 3. This video uses the copyrighted material at issue, but with the appropriate authorization from the copyright owner.

Please explain briefly:

Signature

Jeff Atwood

Type your full name to serve as your electronic signature.

Statement of Good Faith

I have a good faith belief that the material was disabled as a result of a mistake or misidentification, and that I am not intentionally abusing this dispute process.

Type the following statement into the box above

I have a good faith belief that the material was disabled as a result of a mistake or misidentification, and that I am not intentionally abusing this dispute process.

Cancel

Continue

Unfortunately, my fair use claim was denied without explanation by the copyright holder.

Let's consider the four guidelines for fair use I outlined in [my original 2007 blog entry](#):

1. Is the use transformative?
2. Is the source material intended for the public good?
3. How much was taken?
4. What's the market effect?

While we're clear on 3 and 4, items 1 and 2 are hazy in a mashup. This would definitely be transformative, and I like to think that I'm writing for the erudition of myself and others, not merely to entertain people. I uploaded with the intent of the video being viewed through a blog entry,

with YouTube as the content host only. But it was still 90 seconds of the movie viewable on YouTube by anyone, context free.

So I'm torn.

On one hand, this is an insanely impressive technological coup. The idea that YouTube can (with the assistance of the copyright holders) really validate every minute of uploaded video against **every minute of every major copyrighted work** is unfathomable to me. When YouTube promised to do this to placate copyright owners, I was sure they were delaying for time. But much to my fair-use-loving dismay, they've actually gone and built the damn thing—and it works.

Just, maybe, it works a little too well. I'm still looking for video sharing services that [offer some kind of fair use protection](#).

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

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

OceanofPDF.com

VII.

Gaming

OceanofPDF.com

Everything I Needed to Know About Programming I Learned from BASIC

[Edsger Dijkstra](#) had [this](#) to say about Beginner's All Purpose Symbolic Instruction Code:

“It is practically impossible to teach good programming style to students that have had prior exposure to BASIC; as potential programmers they are mentally mutilated beyond hope of regeneration.”

I'm sure he was exaggerating here for effect; as much as I admire his [1972 "The Humble Programmer" paper](#), it's hard to square that humility with the idea that choosing the wrong programming language will damage the programmer's mind. Although [computer languages continue to evolve](#), the largest hurdle I see isn't any particular choice of language, but the fact that [programmers can write FORTRAN in any language](#). To quote Pogo, we have met the enemy, and [he is us](#).

Dismissing BASIC does seem rather elitist. Like many programmers of a certain age, **I grew up with BASIC**.

I mentioned in [an earlier post](#) the curious collision of early console gaming and programming that was the [Atari 2600 BASIC Programming cartridge](#). I had to see this for myself, so I bought a copy on eBay.

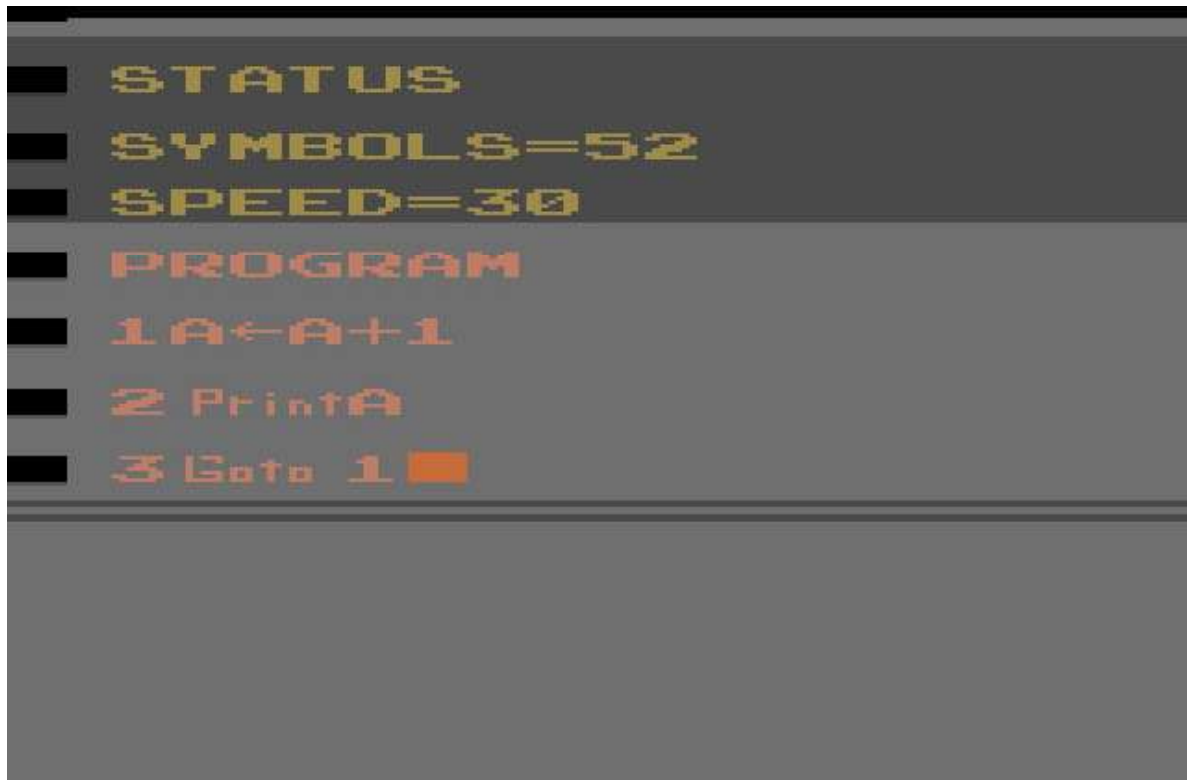


I also bought a set of the Atari 2600 [keypad controllers](#). The overlays come with the cartridge, and the controllers mate together to make a primitive sort of keyboard. (Also, if you were wondering what kinds of things I do with my ad revenue, buying crap like this is a big part of it, sadly.)



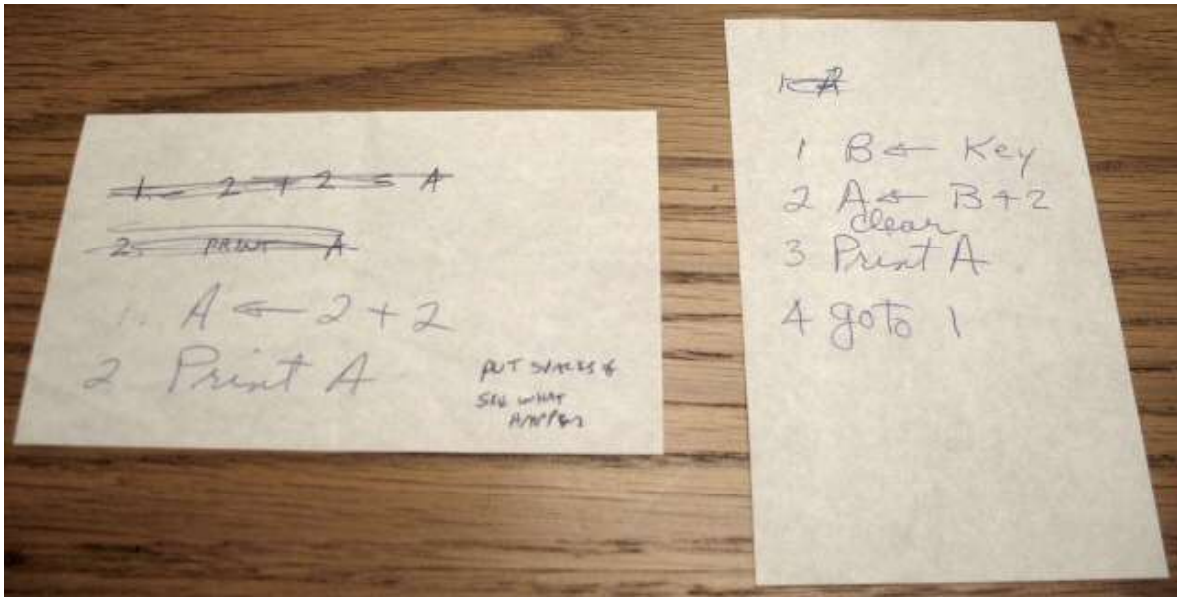
Surprisingly, the manual isn't available anywhere online, so [I scanned it in myself](#). Take a look. It's hilarious. There is a [transcribed HTML version of the manual](#), but it's much less fun to read without the pictures and diagrams.

I booted up a copy of the [Basic Programming ROM](#) in the [Stella Atari 2600 emulator](#), then followed along with the manual and wrote a little BASIC program.



You'll notice that all the other screenshots of Atari 2600 Basic Programming on the web are essentially blank. That's probably because **I'm the only person crazy enough to actually try programming in this thing.** It may look painful, but you have no idea until you've tried to work with this funky "IDE." It's hilariously bad. I could barely stop laughing while punching away at my virtual keypads. But I have to confess, after writing my first "program," I got that same visceral little thrill of bending the machine to my will that I've always gotten.

The package I got from eBay included a few handwritten programming notes that I assume are from the 1980s.



Isn't that what BASIC—even this horribly crippled, elephant man Atari 2600 version of BASIC—is all about? Discovering fundamental programming concepts?

Of course, if you were at all interested in computers, you wouldn't bother programming on a dinky Atari 2600. There were much better options for gaming and programming in the form of home computers. And for the longest time, **every home computer you could buy had BASIC burned into the ROM.** Whether it was the Apple II, Commodore 64, or the Atari 800, you'd boot up to be greeted by a BASIC prompt. It became the native language of the hobbyist programmer.

```
]
]10 PRINT "HELLO WORLD!"
]20 GOTO 10
]RUN*
```

```
READY
10 PRINT "HELLO WORLD!"
20 GOTO 10
RUN■
```

Even the IBM PC had [BASICA](#), [GW-BASIC](#) and finally [QBasic](#), which was phased out with Windows 2000.

It's true that if you wanted to do anything remotely cutting-edge with those old 8-bit Apple, Commodore and Atari home computers, you had to pretty much learn assembly language. I don't recall any compiled languages on the scene until the IBM PC and DOS era, primarily [Turbo Pascal](#). Compiled languages were esoteric and expensive until the great democratization of Turbo Pascal at its low, low price point of \$49.99.

As an aside, you may notice that [Anders Hejlsberg](#) was the primary author of Turbo Pascal and later Delphi; he's now a [Technical Fellow at Microsoft](#) and the chief designer of the C# language. That's a big reason why so many longtime geeks, such as myself, are so gung-ho about .NET.

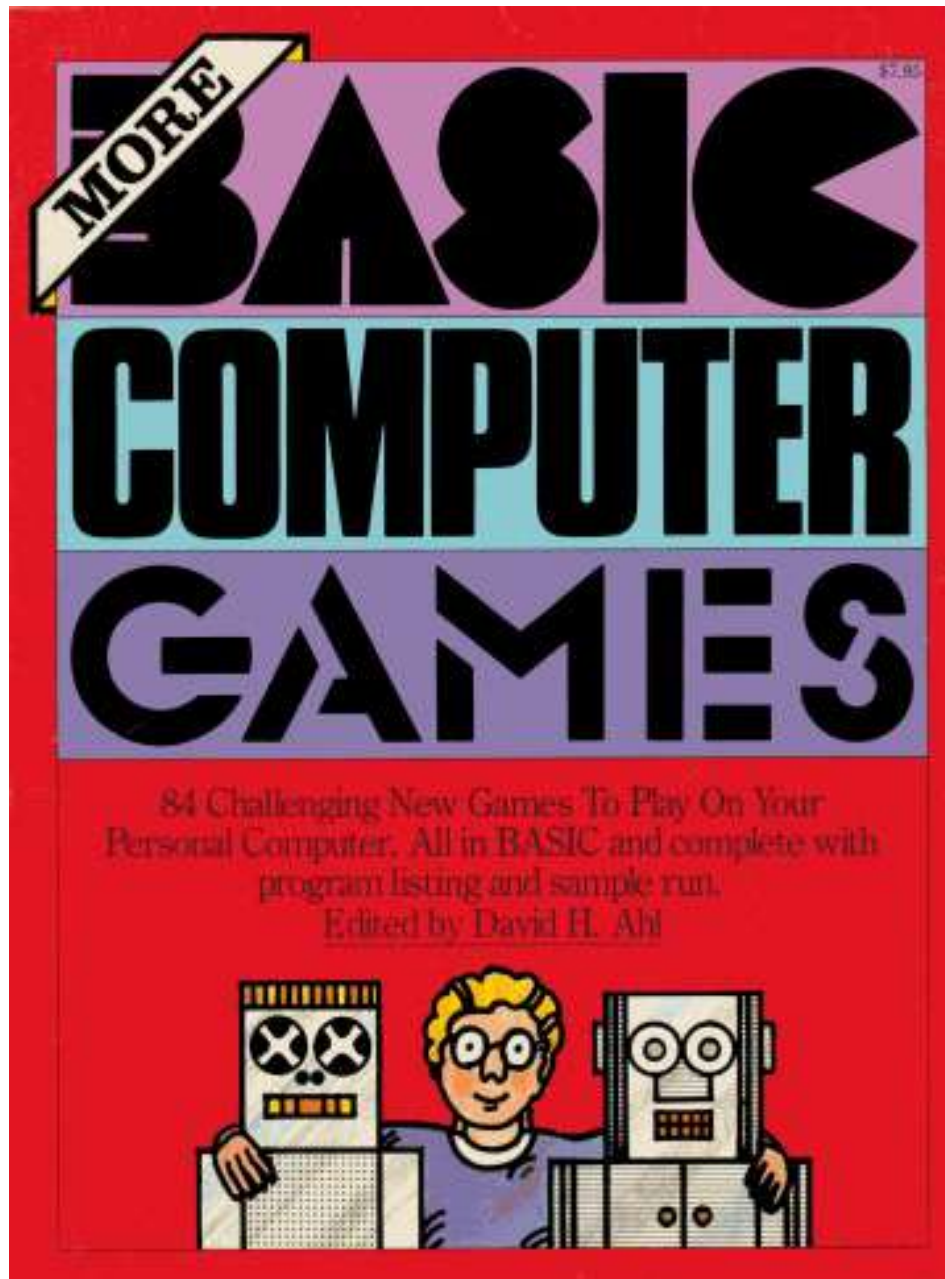
Even if you lacked the programming skills to become the next [David Crane](#) or [Will Wright](#), there were still a lot of interesting games and programs you could still write in good old BASIC. Certainly more than enough to figure out if you enjoyed programming, and if you had any talent. The [Creative Computing compilations](#) were like programming bibles to us.

BASIC COMPUTER GAMES

MICROCOMPUTER EDITION

101 Great Games to Play on Your Home Computer.
By yourself or with others. Each complete with
programming and sample run. Edited by David H. Ahl





For a long, long time, **if you were interested in computers at all, you programmed in BASIC.** It was as unavoidable and inevitable as the air you breathed. Every time you booted up, there was that command prompt blinking away at you. Why not type in some BASIC commands and see what happens? And then the sense of wonder, of possibility, of being able to unlock the infinitely malleable universe inside your computer. Thus the careers of millions of programmers were launched.

BASIC didn't mutilate the mind, as Dijkstra claimed. If anything, BASIC opened the minds of millions of young programmers. It was perhaps the earliest test to determine whether you were [a programming sheep or a non-programming goat](#). Not all will be good, of course, but some inevitably [will go on to be great](#).

Whether we're still programming in it or not, **the spirit of BASIC lives on in all of us.**

OceanofPDF.com

Programming Games, Analyzing Games

For many programmers, **our introduction to programming was our dad forcing us to write our own games.** Instead of the shiny new Atari 2600 game console I wanted, I got a Texas Instruments TI-99/4a computer instead. That's not exactly what I had in mind at the time, of course, but that fateful decision launched a career that spans thirty years.

Evidently, I'm not alone. Mike Lee [had a similar experience](#):

“I was born in 1976, the same year as Apple, so my dad was just the right age to get into the early results of the home-brew movement. One of my few memories of early childhood is of him coming home with a Sinclair 2000 and a book of games. He sat there for hours typing in the code for Space Invaders, and we played it maybe 30 minutes before turning the machine off and undoing all his work.”

As [did Shawn Oster](#):

“I've been developing software for 25 years, since I was 8, starting with a book called ‘Your First BASIC Program’ that my dad bought me because we had a PC while all my friends were playing StarBlazers on their Apple IIs. He said if I wanted to play games, then I could write one myself. At the time I was a bit disappointed (Okay, crushed) but now... well, Dad, thank you.”

That's why it's so fascinating to retrace the earliest computer games. The personal computer industry [grew up with us](#). We learned how to program by [typing in those simple games from magazines and books](#). Look closely, and you'll find that those old game programs are the primitive origins of most programmers, the reptile brain stem we all collectively carry around in our heads.

Even a humble, simple little pack-in game like Minesweeper has deep [roots going back to the days of punch cards](#):

“Minesweeper has its origins in the earliest mainframe games of the '60s and '70s. Wikipedia cites the earliest ancestor of Minesweeper as David Ahl's [Cube](#). But although Cube features ‘landmines,’ it's hard to consider this a predecessor of Minesweeper. In Cube, the mines are placed randomly and the only way to discover where they ends the game. You walk over a landmine and you die; you can't avoid the landmines or know where they are before you take a chance.”



“However, there are a number of very early ‘hide and seek’ games about locating hidden spots on a grid. For example, in Bob Albrecht's [Hurkle](#), you have to find a creature hiding on a ten-by-ten grid. After each guess, you're told in what general direction the Hurkle lies. Dana Nofle's [Depth Charge](#) is the same, but in three dimensions. Bud Valenti's [Mugwump](#) has multiple hidden targets, and after each guess, you get the approximate distance to each of them. Unlike Cube, these games match the general pattern of Minesweeper more closely: make a random guess to start, then start using the information provided by that first guess to uncover the hidden items. Of course, unlike Minesweeper (or Cube), there was no danger of ‘explosion,’ the only constraint was finding the secret locations in a limited number of guesses.

“The closest ancestor to Minesweeper is probably Gregory Yob's [Hunt the Wumpus](#).”

```

EATS NEARBY!
YOU ARE IN ROOM 2
TUNNELS LEAD TO 1 3 10

SHOOT OR MOVE (S-M)?M
WHERE TO?10

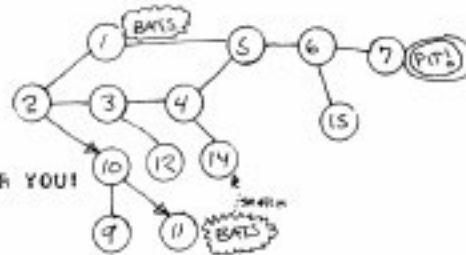
EATS NEARBY!
YOU ARE IN ROOM 10
TUNNELS LEAD TO 2 9 11

SHOOT OR MOVE (S-M)?M
WHERE TO?11
ZAP--SUPER BAT SNATCH! ELSEWHEREVILLE FOR YOU!

YOU ARE IN ROOM 14
TUNNELS LEAD TO 4 13 15

SHOOT OR MOVE (S-M)?M
WHERE TO?15

```

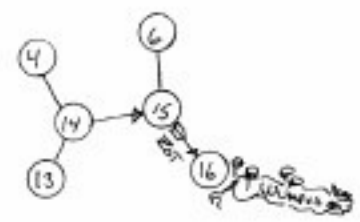


```

I SMELL A WUMPUS!
YOU ARE IN ROOM 15
TUNNELS LEAD TO 6 14 16

SHOOT OR MOVE (S-M)?S
NO. OF ROOMS(1-5)?1
ROOM #?16
AHA! YOU GOT THE WUMPUS!
HEE HEE HEE - THE WUMPUS'LL GETCHA NEXT TIME!!

```



CAN YOU FIT THIS
MAP INTO THE OTHER
ONE ABOVE? FIGURE OUT
HOW I TOOK THE WUMPUS
WAS IN 16.

“Although it used an unorthodox grid (the original game used the vertices of a dodecahedron, and [a later version](#) used Mbius strips and other unlikely patterns), the Wumpus evolved from its predecessors in many other ways.”

I was intrigued by the newfound connection between Minesweeper and [Hunt the Wumpus](#), since [the Wumpus is my power animal](#).

Most of the early games weren't even that much fun. Analyzing the game's program was almost as enjoyable as playing it; the very act of typing it in and understanding the program was "game" enough for many of us. But some of these early games evolved and survived until today, as Minesweeper did—and it has become so ingrained into the public consciousness that it's now the subject of [hilarious parody videos](#). Despite Minesweeper's simplicity (and popularity), it is also a surprisingly deep game of logic, as documented in the [Wikipedia entry](#):

- Analysis: [single square](#), [double square](#), [shared mine](#)
- [NP-Completeness](#)

- [Mine probabilities](#)
- [Measuring Board Difficulty](#)

Minesweeper is still popular with programmers today; [Automine](#), for example, is a Java program that automatically plays Minesweeper by reading the screen and manipulating the mouse.

The Minesweeper article is a part of the amazing [Beyond Tetris series](#) on GameSetWatch, in which many classic puzzle games are examined from the vantage point of a game designer and game programmer. I recommend it highly. Fair warning, though: don't [click through](#) unless you have plenty of time on your hands. For a programmer, **analyzing games is almost as fun as playing them.**

OceanofPDF.com

Game Player, Game Programmer

Greg Costikyan's essay "[Welcome Comrade!](#)" is a call to arms for hobbyist game programmers:



“Back in the day, it took a couple of man days to create a Doom level. Creating a Doom III level took multiple man-weeks. Thus budgets spiral every upward; as late as 1992, a typical computer game had a budget of \$200,000. Today, 10 million dollars is your bare buy-in for a next generation title.

“As budgets soar, publishers are increasingly conservative about what they will fund, because nobody wants to lose 10 million dollars. So they look for ways to reduce their risk. Today, they have become so risk-averse that anything other than a franchise title, a game based on a movie license, or a game that slots easily into a category they know how to sell is unthinkable.

“Today, Myst, Civilization, or Sim City would never get funded.

“We're condemned to more of the same-old same-old from now for all eternity—unless we figure out a way to break this iron grip—what Raph

Koster calls [‘Moore's Wall.’](#)

‘We think it's possible—by building for the game industry what the independent film and independent music movements do for their own industry. Creating a viable ‘independent games’ movement, where people can experiment, at lower budgets and with less risk, on quirky, offbeat, innovative games—and find an audience that prizes gameplay over glitz, innovation over graphical trickery, playfulness over polygons.’

Greg's [Manifesto Games](#) website aims to make this a reality, by creating an audience and supporting hobbyist developers.

James Hague's [Rise and Fall of the Hobbyist Game Programmer](#) documents just how profoundly the world has changed for would-be game programmers in the last 30 years:

“For a small percentage of enthusiasts, there's always been the calling to jump from just playing games to creating them. It's crazy, of course, because the rush of playing a great game doesn't carry over to spending twenty straight hours in the basement trying to figure out why a level initialization routine fails ten percent of the time. But those that persisted, they drove the industry in its early days.

“I remember reading about Mark Turmell—and others whose names I've forgotten—who were somehow inspired to design their own games, and then sit down and figure out exactly how to turn them into something their friends could actually come over and play. Those were fantastic feats that started the chatter about computer games becoming a new art form. One person, one vision, and six months later a finished product that was snapped up by a publisher—pure creation. A new alternative for would-be novelists.

“The dream is still alive in these days of 32-bit processors and 3D accelerators, but over the years the reality behind it has quietly slipped away and few have stopped to notice.

“In 1981, personal computers were in the thick of their 8-bit heyday. Not only are we talking about an 8-bit 6502—a processor with one primary register and no multiply instruction—running at less than 2 megahertz, but it was still acceptable, though just barely, to write games in BASIC. Now don't get me wrong, BASIC was the downtrodden interpreted language that

it still is, but it shipped with every Apple II and Atari 800, and was the obvious choice for budding programmers.”

Perhaps this is another reason why [the Visual Studio Express IDE should ship with Vista](#). Or maybe doing it as-is with the .NET 2.0 command-line compiler and Notepad is more authentic. For more perspective on how the game programming world has changed since those early days, I can highly recommend James Hague's essential 1997 e-book [Halcyon Days: Interviews with Classic Computer and Video Game Programmers](#).

Perhaps it's a little easier to imagine transitioning from gamer to programmer in the world of mobile devices. [Lightworks games](#) is doing just that—two guys pursuing their dream by building a game company from the ground up on the PocketPC. They started with [Cavemen](#), a charming little [Lemmings clone](#).

There's also Microsoft's intriguing [XNA Game Studio](#), which is [now available in beta](#). It's a way for hobbyist developers to write non-commercial games that run on the Xbox 360. There have been rumblings about the best of these non-commercial games eventually making their way to the Xbox Live marketplace—which could potentially convert those hobbyist game programmers into small business owners. It's an exciting prospect, given the huge installed base of most consoles, and the ease of getting everything running on a standard console platform.

Is the dream of jumping from game player to game programmer still alive? It's certainly how I got my start in programming.

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

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

[OceanofPDF.com](#)

VIII.

Things to Read

OceanofPDF.com

Programmers Don't Read Books, But You Should

One of the central themes of stackoverflow.com is that software developers no longer learn programming from books, as [Joel mentioned](#):

“Programmers seem to have stopped reading books. The market for books on programming topics is miniscule compared to the number of working programmers.”

Joel expressed similar sentiments in 2004's [The Shlemiel Way of Software](#):

“But the majority of people still don't read. Or write. The majority of developers don't read books about software development, they don't read Web sites about software development, they don't even read Slashdot.”

If programmers don't learn from books today, how do they learn to program? They do it the old-fashioned way: by rolling up their sleeves and writing code— while harnessing the collective wisdom of the internet in a second window. The internet has rendered programming books obsolete. It's faster, more efficient, and just plain smarter to get your programming information online. I believe Doug McCune's experience, which he aptly describes as [Why I Don't Read Books](#), is fairly typical.

I lay part of the blame squarely at the feet of the technical book publishing industry:

1. **Most programming books suck.** The barrier to being a book author, as near as I can tell, is virtually nonexistent. The signal to noise of book publishing is arguably not a heck of a lot better than what you'll find on the wilds of the internet. Of the hundreds of programming books released every year, perhaps two are three are truly worth the time investment.
2. **Programming books sold by weight, not by volume.** There seems to be an inverse relationship between the size of a programming book and its quality. The bigger the book, somehow, the less useful information it

will contain. What is the point of these giant wanna-be reference tomes? How do you find anything in it, much less lift the damn things?

3. **Quick-fix programming books oriented towards novices.** I have nothing against novices entering the programming field. But I continue to believe the "Learn [Insert Language Here] in 24 hours!" variety of books are [doing our profession a disservice](#). The monomaniacal focus on right now and the fastest, easiest possible way to do things leads beginners down the wrong path—or as I like to call it, "PHP." I kid! I kid!
4. **Programming book pornography.** The idea that having a pile of thick, important-looking programming books sitting on your shelf, largely unread, will somehow make you a better programmer. As [David Poole](#) once related to me in email, "I'd never get to do that in real life" seems to be the theme of the programming book porn pile. This is why I considered, and rejected, buying Knuth's [Art of Computer Programming](#). Try to purchase practical books you'll actually read, and more importantly, put into action.

As an author, I'm guilty, too. I co-wrote a programming book, and [I still don't think you should buy it](#). I don't mean that in an ironic-trucker-hat, reverse-psychology way. I mean it quite literally. It's not a bad book by any means. I have the utmost respect for my [esteemed co-authors](#). But the same information would be far more accessible on the web. Trapping it inside a dead tree book is ultimately a waste of effort.

The internet has certainly accelerated the demise of programming books, but there is some evidence that, even pre-internet, programmers didn't read all that many programming books. I was quite surprised to encounter the following passage in "[Code Complete](#):"

"Pat yourself on the back for reading this book. You're already learning more than most people in the software industry because one book is more than most programmers read each year (DeMarco and Lister 1999). A little reading goes a long way toward professional advancement. If you read even one good programming book every two months, roughly 35 pages a week, you'll soon have a firm grasp on the industry and distinguish yourself from nearly everyone around you."

I believe the same text is present in the original 1993 edition of Code Complete, but I no longer have a copy to verify that. A little searching uncovered the passage Steve McConnell is referencing in DeMarco and Lister's "[Peopleware](#):"

“The statistics about reading are particularly discouraging: The average software developer, for example, doesn't own a single book on the subject of his or her work, and hasn't ever read one. That fact is horrifying for anyone concerned about the quality of work in the field; for folks like us who write books, it is positively tragic.”

It pains me greatly to [read the reddit comments](#) and learn that people are interpreting the stackoverflow.com mission statement as a repudiation of programming books. As ambivalent as I am about the current programming book market, **I love programming books!** This very blog was founded on the concept of my [recommended developer reading list](#). Many of my blog posts are my [feeble attempts to explain key concepts](#) outlined long ago in classic programming books.

How to reconcile this seemingly contradictory statement, the [love and hate dynamic](#)? You see, there are programming books, and there are programming books.

The best programming books are timeless. They transcend choice of language, IDE, or platform. They do not explain how, but why. If you feel compelled to clean house on your bookshelf every five years, trust me on this, **you're buying the wrong programming books.**

I wouldn't trade my programming bookshelf for anything. I refer to it all the time. In fact, I referred to it twice while composing this very post.



I won't belabor my [recommended reading list](#), as I've kept it proudly the same for years.

But I do have this call to arms: **my top five programming books every working programmer should own—and read.** These seminal books are richly practical reads, year after year, no matter what kind of programming I'm doing. They reward repeated readings, offering deeper and more penetrating insights into software engineering every time I return to them, armed with a few more years of experience under my belt. If you haven't read these books, what are you waiting for?

- [Code Complete 2](#)
- [Don't Make Me Think](#)

- [Peopleware](#)
- [Pragmatic Programmer](#)
- [Facts and Fallacies](#)

It is my greatest intention to make [stackoverflow.com](#) highly complementary to these sorts of timeless, classic programming books. It is in no way, shape, or form meant as a replacement for them.

On the other hand, if you're the unfortunate author of "[Perl for Dummies](#)," then watch your back, because we're definitely gunning for you.

[OceanofPDF.com](#)

Nobody's Going to Help You, and That's Awesome

I'm not into [self-help](#). I don't buy self-help books, I don't read productivity blogs, and I certainly don't subscribe to [self-proclaimed self-help guru](#) newsletters. Reading someone else's advice on the rather generic concept of helping yourself always struck me as a particularly misguided idea.

Apparently I'm not the only person to [reach this conclusion](#), either.

“I spent two years reading all the self-help books I could find. As of a year ago, I had read 340 self-help books. Because I'm insane.

“My conclusion from all that reading?

“95 percent of self-help books are complete bullshit.”

To be clear, I am all for self-improvement. Reading books, blogs, and newsletters by people who have accomplished great things is a fine way to research your own path in life. But these people, however famous and important they may be, [are not going to help you](#).

“Unfortunately that's not the answer he wanted. To him, my answer [that nobody was going to help him become successful] was really discouraging. To me, if I was receiving that answer from someone else, it would be really encouraging.

“I like being reminded that nobody's going to help me—that it's all up to me. It puts my focus back on the things I can control—not waiting for outside circumstances.”

Take it from [The Ultimate Productivity Blog](#):

You should be working.

Reading self-help advice from other people, however well-intentioned, is no substitute for **getting your own damn work done**. The sooner you come to terms with this, the better off you'll be.

Get out there and do stuff because you fundamentally enjoy it and because it makes you better. As a writer, as an analyst, as a techie, whatever. Learn to [love practicing the fundamentals](#) and **do it better each time**. Over time, quality does lead to success, but you have to be patient. Really patient. Turns out, ["overnight" success takes years](#). Maybe even decades. This is not a sprint, it's a marathon. Plan accordingly.

For example, I don't care if anyone reads what I write here. I'm writing to satisfy myself first and foremost. If others read it and benefit from it, fantastic—that's a welcome side effect. If I worry about who is reading, why they're reading, or if anyone is even reading at all, I'd be too paralyzed to write! That'd be the least productive outcome of all.

That's not to say that some introspection about the nature of your work isn't useful. It is. Even [the weary self-help student](#) I quoted above concluded that five percent of self-help advice surprisingly wasn't bullshit. The one book he recommended without hesitation? [59 Seconds: Think a Little, Change a Lot](#).”



Despite my deep reservations about the genre, I ordered this book based on his recommendation and a number of credible references to it I noticed on the [Skeptic Stack Exchange](#).

Why does this self-help book work when so many others fail? In a word, **science!** The author goes out of his way to find actual published scientific research documenting specific ways we can make small changes in our behavior to produce better outcomes for ourselves and those around us. It's powerful stuff, and the book is full of great, research-backed insights like this one:

“A group of participants were asked to select a negative experience. One group of participants were then asked to have a long chat with a supportive experimenter about the event, while a second group were invited to chat about a far more mundane topic—a typical day.

“Participants who had spent time talking about their traumatic event through the chat had been helpful. However, the various questionnaires told

a very different story. In reality the chat had no significant impact at all. They might just as well have been chatting about a typical day.

“In several studies, participants who have experienced a traumatic event have been encouraged to spend just a few minutes each day writing in a diary-type account of their deepest thoughts and feelings about it. For example, in one study participants who had just been made redundant were asked to reflect upon their deepest thoughts and feelings about their job loss, including how it had affected both their personal and professional lives. Although these types of exercises were both speedy and simple, the results revealed a remarkable boost in their psychological and physical well-being, including a reduction in health problems and an increase in self-esteem and happiness.

“The results left psychologists with something of a mystery. Why would talking about a traumatic experience have almost no effect but writing about it yield such significant benefits? From a psychological perspective, talking and writing are very different. Talking can often be somewhat unstructured, disorganized, even chaotic. In contrast, writing encourages the creation of a story line and structure that help people make sense of what has happened and work towards a solution. In short, talking can add to a sense of confusion, but writing provides a more systematic, solution-based approach.”

Therefore, the real world change you would make based on this advice—the proverbial 59 seconds on the book jacket—is to avoid talking through traumatic experiences in favor of writing about them. Not because some self-help guru said so, but because the published research data tells us that talking doesn't work and writing does. Not exactly intuitive, since talking through our problems with a friend always feels like the right thing to do, but I have certainly documented many times over the value of writing through a problem.

[59 Seconds](#) is so good, in fact, **it has rekindled my hopes that our new [Stack Exchange Productivity Q&A](#) can work.** I'd love for our productivity site to be founded on a scientific basis, and not the blind cult of personality I've come to expect from the self-help industry.

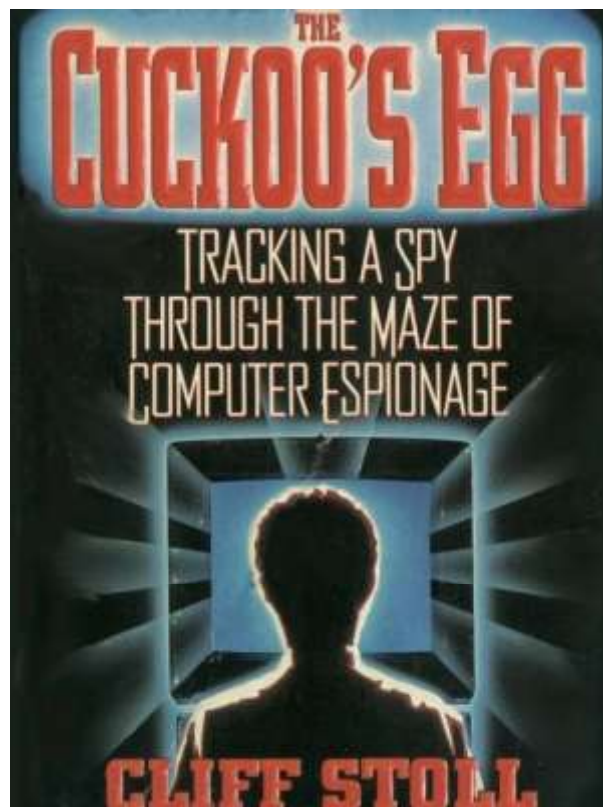
Remember, nobody's going to help you ... except science, and if you're willing to put in the required elbow grease each and every day—yourself.

OceanofPDF.com

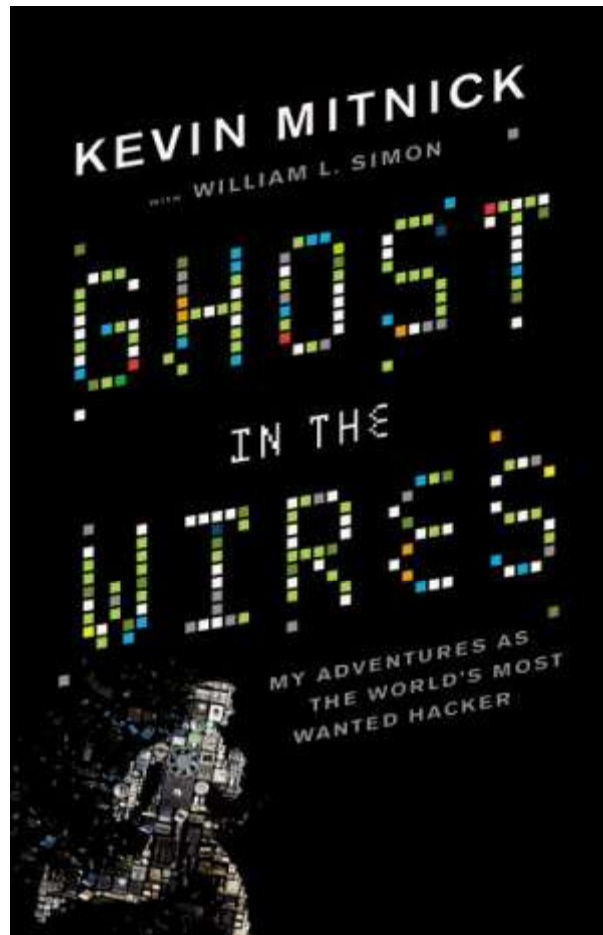
Computer Crime, Then and Now

I've already [documented](#) my brief, youthful dalliance with the illegal side of computing as it existed in the late 1980s. But was it crime? Was I truly a criminal? I don't think so. To be perfectly blunt, I wasn't talented enough to be any kind of threat. I'm still not.

There are two classic books describing hackers active in the 1980s who did have incredible talent. Talents that made them dangerous enough to be considered criminal threats.



[The Cuckoo's Egg: Tracking a Spy Through the Maze of Computer Espionage](#)



[Ghost in the Wires: My Adventures as the World's Most Wanted Hacker](#)

Cuckoo is arguably the first case of hacking that was a clearly malicious crime circa 1986, and certainly the first known case of computer hacking as international espionage. I read this when it was originally published in 1989, and it's still a gripping investigative story. Cliff Stoll is a visionary writer who saw **how trust in computers and the emerging Internet could be vulnerable to real, actual, honest-to-God criminals.**

I'm not sure [Kevin Mitnick](#) did anything all that illegal, but there's no denying that he was the world's first high profile computer criminal.

U.S. Department of Justice
United States Marshals Service

WANTED BY U.S. MARSHALS

NOTICE TO ARRESTING AGENCY: Before arrest, validate warrant through National Crime Information Center (NCIC).
United States Marshals Service NCIC entry number: (NCIC/ W721460021).

NAME:MITNICK, KEVIN DAVID
AKS (S):MITNICK, KEVIN DAVID
 MERRILL, BRIAN ALLEN

DESCRIPTION:

Sex:MALE
Race:WHITE
Place of Birth:VAN NUTS, CALIFORNIA
Date(s) of Birth:08/06/63; 10/18/70
Height:5'11"
Weight:190
Eyes:BLUE
Hair:BROWN
Skin tone:LIGHT
Scars, Marks, Tattoos:NONE KNOWN
Social Security Number (s):550-39-5695
NCIC Fingerprint Classification: ...DOPM2OPM13DIPM19PM09



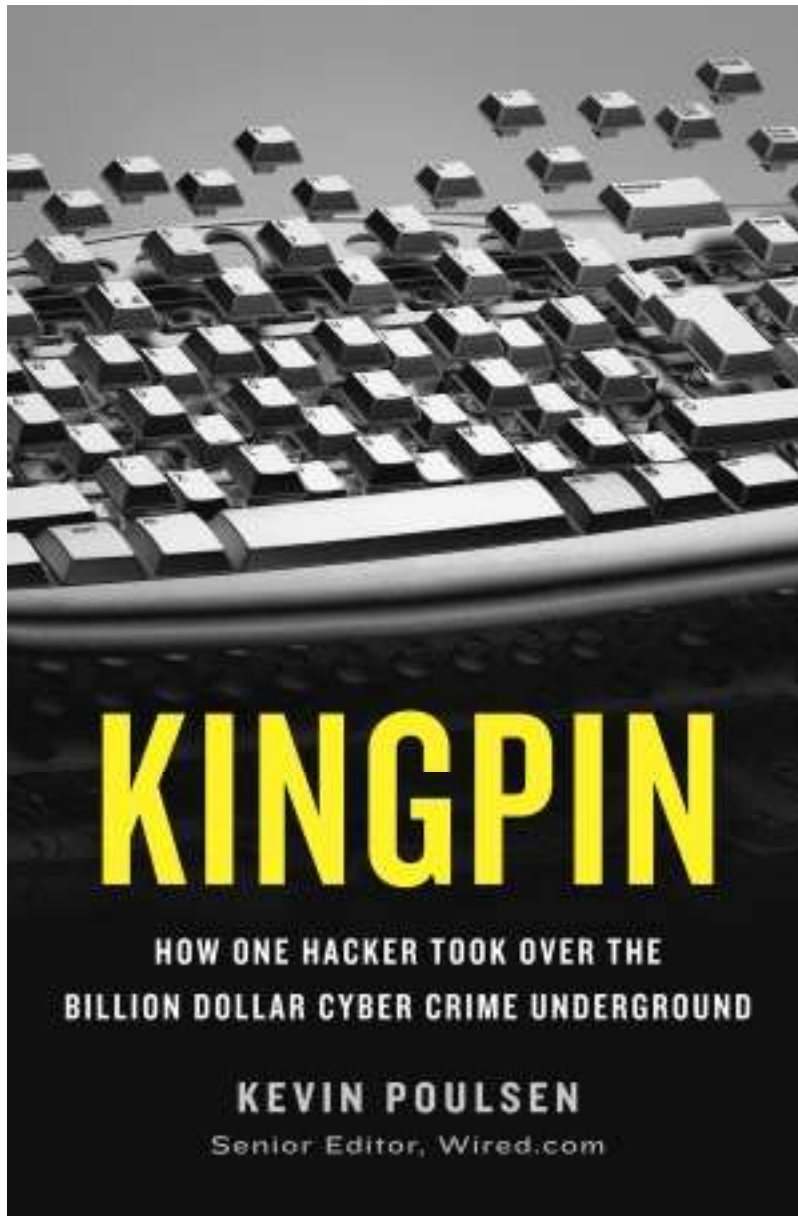
ADDRESS AND LOCALE: KNOWN TO RESIDE IN THE SAN FERNANDO VALLEY AREA OF CALIFORNIA AND LAS VEGAS, NEVADA

By 1994, he made the FBI's 10 Most Wanted list, and there were [front page New York Times articles about his pursuit](#). If there was ever a moment that computer crime and "hacking" entered the public consciousness as an ongoing concern, this was it.

The whole story is told in minute detail by Kevin himself in [Ghost in the Wires](#). There was a sanitized version of Kevin's story presented in [Wizywig comix](#) but this is the original directly from the source, and it's well worth reading. I could barely put it down. Kevin has been fully reformed for many years now; he wrote [several books](#) documenting his techniques and now [consults](#) with companies to help improve their computer security.

These two books cover the genesis of all computer crime as we know it. Of course it's a much bigger problem now than it was in 1985, if for no other reason than there are far more computers far more interconnected with each other today than anyone could have possibly imagined in those early days. But what's really surprising is **how little has changed in the techniques of computer crime since 1985.**

The best primer of modern—and by that I mean year 2000 and later—computer crime is [Kingpin: How One Hacker Took Over the Billion-Dollar Cybercrime Underground](#). Modern computer crime is more like the classic sort of crime you've seen in black and white movies: it's mostly about stealing large sums of money. But instead of busting it out of bank vaults Bonnie and Clyde style, it's now done electronically, mostly through ATM and credit card exploits.



Written by [Kevin Poulsen](#), another famous reformed hacker, “[Kingpin](#)” is also a compelling read. I've read it twice now. The passage I found most

revealing is this one, written after the protagonist's release from prison in 2002:

“One of Max’s former clients in Silicon Valley tried to help by giving Max a \$5,000 contract to perform a penetration test on the company’s network. The company liked Max and didn’t really care if he produced a report, but the hacker took the gig seriously. He bashed at the company’s firewalls for months, expecting one of the easy victories to which he’d grown accustomed as a white hat. But he was in for a surprise. The state of corporate security had improved while he was in the joint. He couldn’t make a dent in the network of his only client. His 100 percent success record was cracking.

“Max pushed harder, only becoming more frustrated over his powerlessness. Finally, he tried something new. Instead of looking for vulnerabilities in the company’s hardened servers, he targeted some of the employees individually.

“These ‘client side’ attacks are what most people experience of hackers—a spam e-mail arrives in your in-box, with a link to what purports to be an electronic greeting card or a funny picture. The download is actually an executable program, and if you ignore the warning message.”

All true; no hacker today would bother with frontal assaults. The chance of success is miniscule. Instead, they target the soft, creamy underbelly of all companies: the users inside. Max, the hacker described in *Kingpin*, bragged, “I’ve been confident of my 100 percent [success] rate ever since.” This is the new face of hacking. Or is it?

One of the most striking things about [Ghost In The Wires](#) is not how skilled a computer hacker Kevin Mitnick is (although he is undeniably great), but how devastatingly effective he is at **tricking people into revealing critical information in casual conversations**. Over and over again, in hundreds of subtle and clever ways. Whether it's 1985 or 2005, the amount of military-grade security you have on your computer systems matters not at all when someone using those computers [clicks on the dancing bunny](#). Social engineering is [the most reliable and evergreen hacking technique ever devised](#). It will outlive us all.

For a 2012 era example, consider [the story of Mat Honan](#). It is not unique.

“At 4:50 PM, someone got into my iCloud account, reset the password and sent the confirmation message about the reset to the trash. My password was a 7 digit alphanumeric that I didn’t use elsewhere. When I set it up, years and years ago, that seemed pretty secure at the time. But it’s not. Especially given that I’ve been using it for, well, years and years. My guess is they used brute force to get the password and then reset it to do the damage to my devices.”

I heard about this on Twitter when the story was originally developing, and my initial reaction was skepticism that anyone had bothered to brute force anything at all, since [brute forcing is for dummies](#). Guess what it turned out to be. Go ahead, guess!

Did you by any chance guess [social engineering... of the account recovery process](#)? Bingo.

“After coming across my [Twitter] account, the hackers did some background research. My Twitter account linked to my personal website, where they found my Gmail address. Guessing that this was also the e-mail address I used for Twitter, Phobia went to Google’s account recovery page. He didn’t even have to actually attempt a recovery. This was just a recon mission.

“Because I didn’t have Google’s two-factor authentication turned on, when Phobia entered my Gmail address, he could view the alternate e-mail I had set up for account recovery. Google partially obscures that information, starring out many characters, but there were enough characters available, m••••n@me.com. Jackpot.

“Since he already had the e-mail, all he needed was my billing address and the last four digits of my credit card number to have Apple’s tech support issue him the keys to my account.

“So how did he get this vital information? He began with the easy one. He got the billing address by doing a who-is search on my personal web domain. If someone doesn’t have a domain, you can also look up his or her information on Spokeo, WhitePages, and PeopleSmart.

“Getting a credit card number is trickier, but it also relies on taking advantage of a company’s back-end systems... First you call Amazon and

tell them you are the account holder, and want to add a credit card number to the account. All you need is the name on the account, an associated e-mail address, and the billing address. Amazon then allows you to input a new credit card. (Wired used a bogus credit card number from a website that generates fake card numbers that conform with the industry's published self-check algorithm.) Then you hang up.

“Next you call back, and tell Amazon that you've lost access to your account. Upon providing a name, billing address, and the new credit card number you gave the company on the prior call, Amazon will allow you to add a new e-mail address to the account. From here, you go to the Amazon website, and send a password reset to the new e-mail account. This allows you to see all the credit cards on file for the account—not the complete numbers, just the last four digits. But, as we know, Apple only needs those last four digits.”

Phobia, the hacker Mat Honan documents, was a minor who did this for laughs. One of his friends is [a 15 year old hacker who goes by the name of Cosmo](#); he's the one who discovered the Amazon credit card technique described above. And what are teenage hackers up to these days?

“Xbox gamers know each other by their gamertags. And among young gamers it's a lot cooler to have a simple gamertag like 'Fred' than, say, 'Fred1988Ohio.' Before Microsoft beefed up its security, getting a password-reset form on Windows Live (and thus hijacking a gamer tag) required only the name on the account and the last four digits and expiration date of the credit card on file. Derek discovered that the person who owned the 'Cosmo' gamer tag also had a Netflix account. And that's how he became Cosmo.

“I called Netflix and it was so easy,' he chuckles. 'They said, 'What's your name?' and I said, 'Todd [Redacted],’ gave them his e-mail, and they said, 'Alright your password is 12345,' and I was signed in. I saw the last four digits of his credit card. That's when I filled out the Windows Live password-reset form, which just required the first name and last name of the credit card holder, the last four digits, and the expiration date.

“This method still works. When Wired called Netflix, all we had to provide was the name and e-mail address on the account, and we were given the

same password reset.”

The techniques are [eerily similar](#). The only difference between Cosmo and Kevin Mitnick is that they were born in different decades. Computer crime is a whole new world now, but the techniques used today are almost identical to those used in the 1980s. If you want to engage in computer crime, don't waste your time developing ninja level hacking skills, **because computers are not the weak point.**

People are.

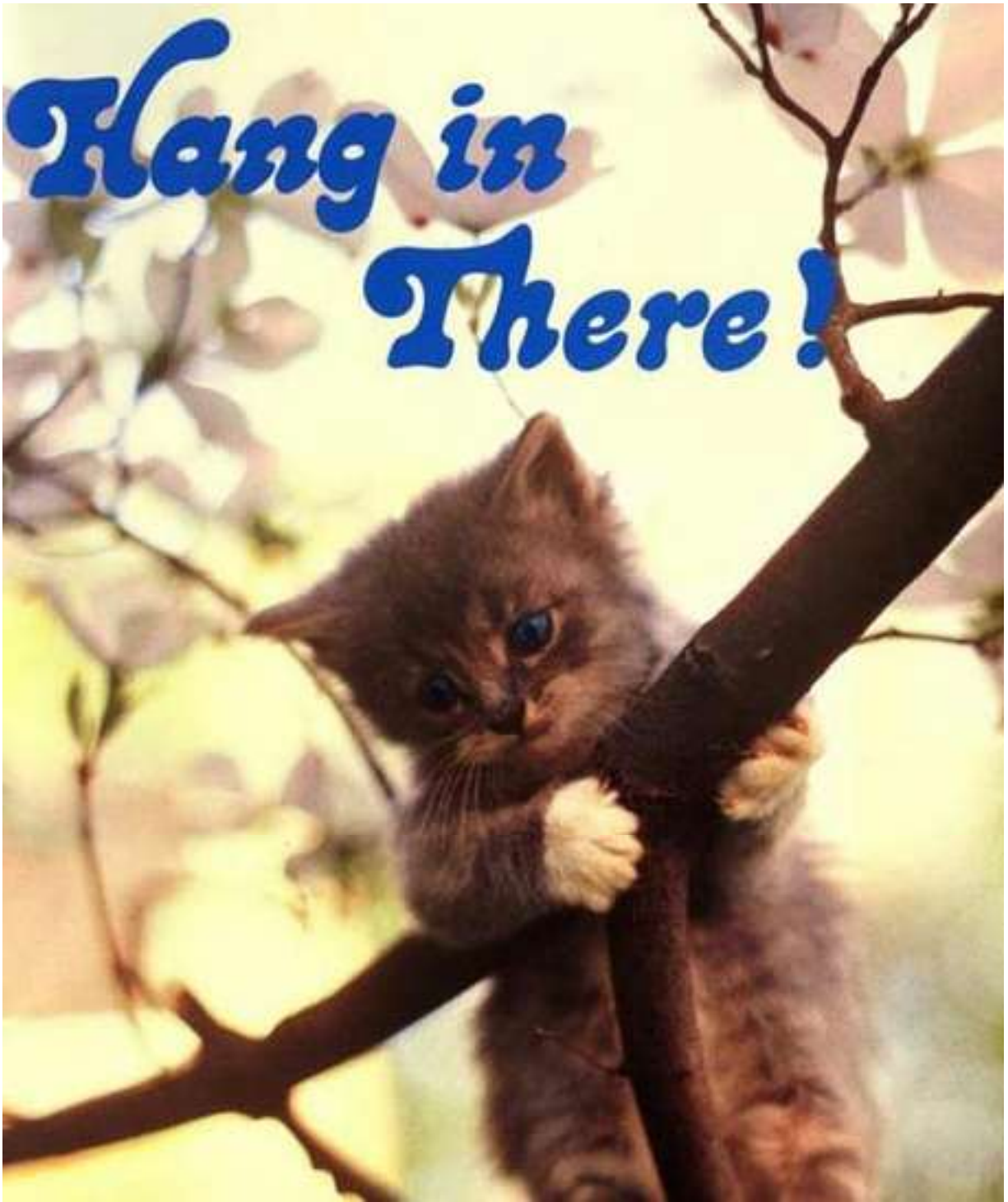
OceanofPDF.com

How to Talk to Human Beings

I hesitate to say everyone should have a child, because becoming a parent is an intensely personal choice. I try my best to avoid evangelizing the experience, but the deeper in I get, the more I believe that **nothing captures the continued absurdity of the human condition better than having a child does.**

After becoming a parent, the first thing you'll say to yourself is, my God, it is a miracle any of us even exist, because I want to freakin' kill this kid at least three times a day. But then your child will spontaneously hug you, or tell you some stupid joke that they can't stop laughing at, or grab for your hand while crossing the street and then ... well, here we all are, aren't we? I'm left wondering if I'll ever be able to love other people—or for that matter myself—as much as I love my children. Unconditional, irrational, nonsensical love. That's humanity in a nutshell.

Parenting is by far the toughest job I've ever had. It makes my so-called career seem awfully quaint in comparison.



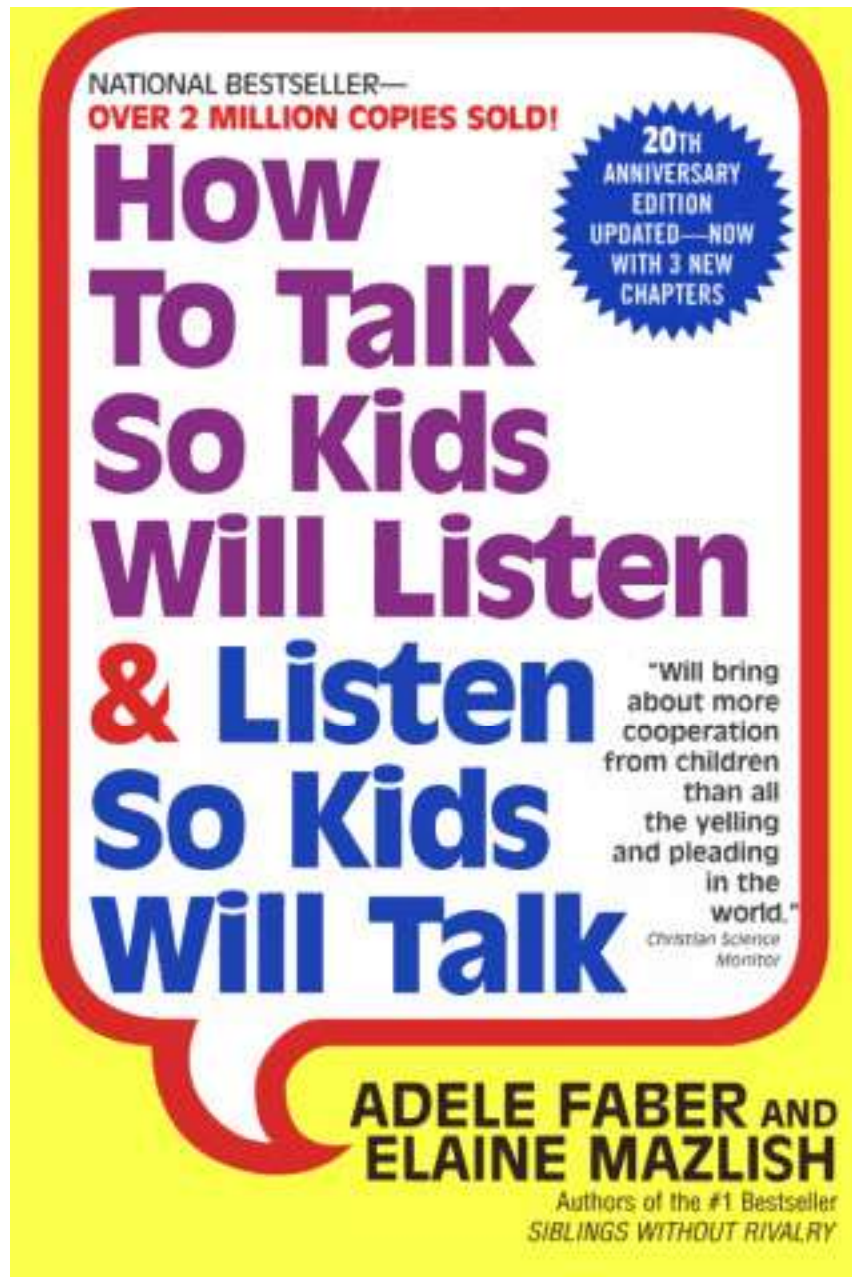
My favorite part of the parenting process, though, is finally being able to talk to my kids. When the dam breaks and all that crazy stuff they had locked away in those tiny brains for the first two years comes uncontrollably pouring out. Finding out what they're thinking about and what kind of people they are at last. Watching them discover and explore

the surface of language is utterly fascinating. After spending two years trying to guess—with extremely limited success—what they want and need, truly, what greater privilege is there than to simply ask them? Language: [Best. Invention. Ever.](#) I like it so much I'm using it right now!

Language also allows kids to demonstrate just what crazy little rolling balls of id they (and by extension, we) all are on the inside. Kids don't know what it means to be mad, to be happy, to be sad. They have to be taught what emotions are, how to handle them, and how to deal in a constructive way with everything the world is throwing at them. You'll get a ringside seat to this process not as a passive observer, but as their coach and spirit guide. They have no coping mechanisms except the ones we teach them. **The difference between a child who freaks out at the slightest breeze, and a child who can confidently navigate an unfamiliar world?** The parents.

See, I told you this was going to be tough.

There are of course innumerable books on parenting and child-rearing, most of which I have no time to read because by the time I'm done being a parent for the day, I'm too exhausted to read more about it. And, really, who wants to read about parenting when you're living the stuff 24/7? Except on [Parenting Stack Exchange](#), of course. However, there is one particular book I happened to discover that was shockingly helpful, even after barely ten pages in. If you ever need to deal with children aged 2 to 99, **stop reading right now and go buy** [“How to Talk So Kids Will Listen and Listen So Kids Will Talk.”](#)



We already own three copies. And you're welcome.

What's so great about this book? I originally found it through A.J. Jacobs, who I mentioned in [Trust Me, I'm Lying](#). Here's how he describes it:

The best marriage advice book I've read is a paperback called How to Talk So Kids Will Listen and Listen So Kids Will Talk. As you might deduce from the title, it wasn't meant as a marriage advice book. But the techniques in this book are so brilliant, I use them in every human interaction I can, no

matter the age of the conversant. It's a strategy that was working well until today.

The book was written by a pair of former New York City teachers, and their thesis is that we talk to kids all wrong. You can't argue with kids, and you shouldn't dismiss their complaints. The magic formula includes: listen, repeat what they say, label their emotions. The kids will figure out the solution themselves.

I started using it on Jasper, who would throw a tantrum about his brothers monopolizing the pieces to Mouse Trap. I listened, repeated what he said, and watched the screaming and tears magically subside. It worked so well, I decided, why limit it to kids? My first time trying it on a grown-up was one morning at the deli. I was standing behind a guy who was trying unsuccessfully to make a call on his cell.

'Oh come on! I can't get a signal here? Dammit. This is New York.'

He looked at me.

'No signal?' I say. 'Here in New York?' (Repeat what they say.)

'It's not like we're in goddamn Wisconsin.'

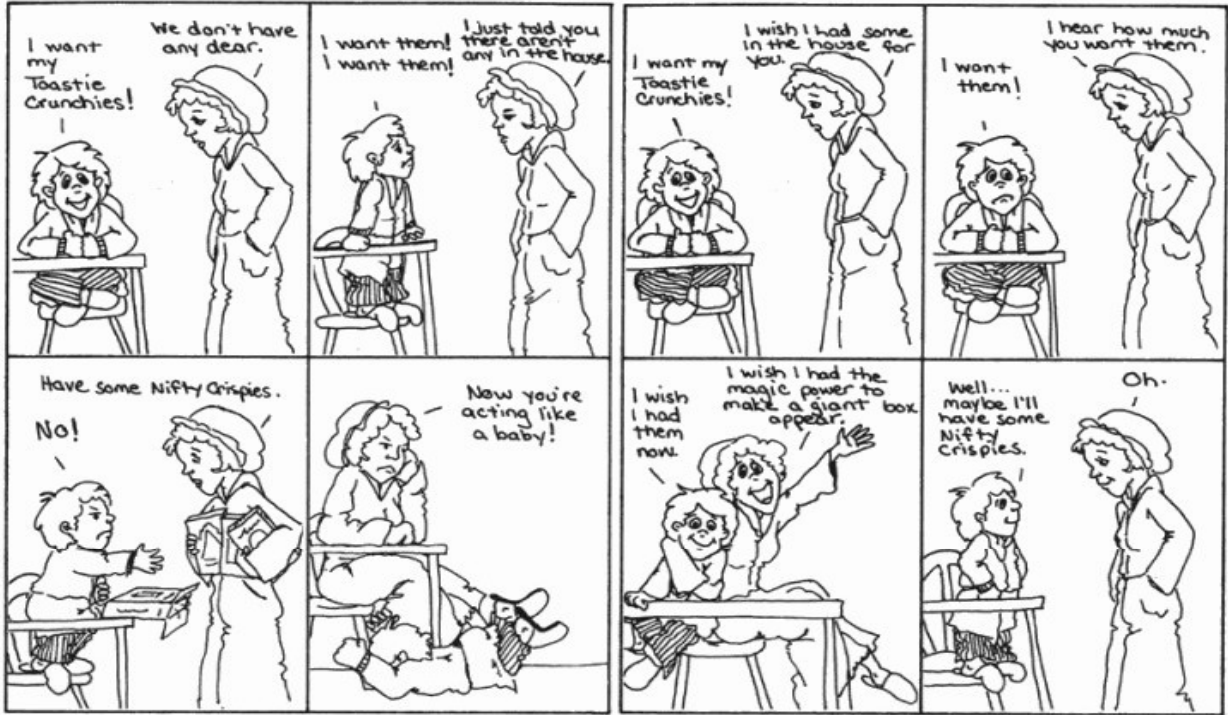
'Mmmm.' (Listen. Make soothing noises.)

'We're not on a farm. It's New York, for God's sake,' he said.

'That's frustrating,' I say. (Label their emotions.)

He calmed down."

This book taught me that, as with so many other things in life, I've been doing it all wrong. I thought it was my job as a parent to solve problems for my children, to throw myself on life's figurative grenades to protect them. Consider the following illustrated examples from the book.



Notice how she cleverly lets the child reach an alternative solution himself, rather than providing the "solution" to him on a silver platter as the all-seeing, all-knowing omniscient adult. This honestly would never have occurred to me, because, well, if we're out of Toastie Crunchies, then we are out of freaking Toastie Crunchies!



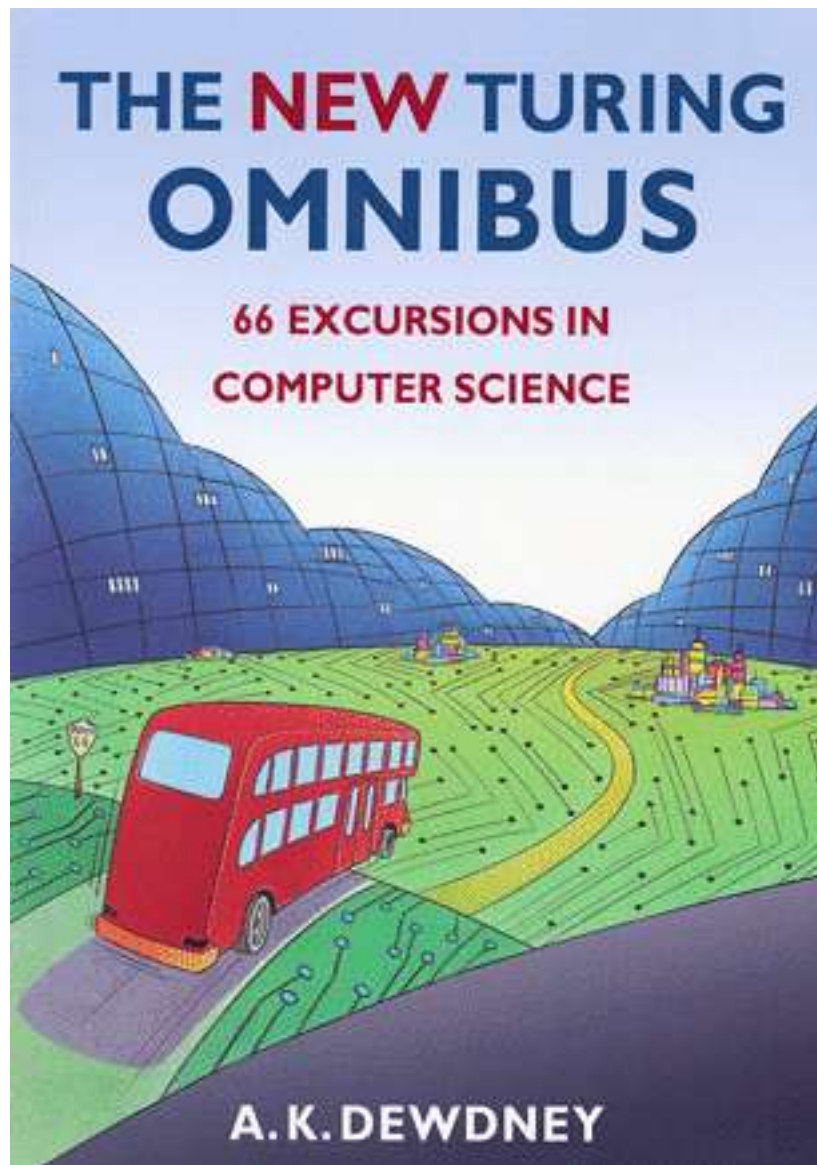
I've learned to fall back whenever possible to simply describing things or situations instead of judging or pontificating. I explain the consequences of potential actions rather than jumping impatiently to "don't do that."

[How to Talk So Kids Will Listen & Listen So Kids Will Talk](#) is full of beautiful little insights on human interaction like this, and I was surprised to find how often what I thought was a good parenting behavior was working against us. Turns out, children aren't the only ones who have trouble dealing with their emotions and learning to communicate. I haven't just improved my relationship with my kids using the practical advice in this book, **I've improved my interactions with all human beings from age 2 to 99.**

Kids will teach you, if you let them. They'll teach you that getting born is the easy part. Anyone can do that in a day. But becoming a well-adjusted human being? That'll take the rest of your life.

Practicing the Fundamentals: The New Turing Omnibus

While researching [Classic Computer Science Puzzles](#), our CEO Scott Stanfield turned me on to A.K. Dewdney's "[The New Turing Omnibus: 66 Excursions in Computer Science](#)."



This is an incredibly fun little book. Sure, it's got Towers of Hanoi, but it's also got so much more:

“The book is designed to appeal both to the educated layperson and to the student of computer science. But how is that possible? The answer lies in the variety of treatments as well as topics. Some of the topics are inherently easy or I have been lucky enough to stumble upon just the right expository mechanisms. Some of the topics are inherently deep or complicated and there is no way around a certain rigor, including occasional mathematical symbolism.

“For students of computer science, the 66 chapters that follow will give a sneak preview of the major ideas and techniques they will encounter in their undergraduate careers and some they may only encounter as graduate students. For professors of computer science, my colleagues, the 66 chapters will amount to a sneak review. Trying to remember how the Boyer-Moore string-matching algorithm went? It's right there in Chapter 61, Searching Strings. As for your lectures, if you like to deliver your own material this book may be what you've been looking for.

“At one end of its spectrum of uses, “[The \(New\) Turing Omnibus](#)” may be ideal in bringing students from diverse backgrounds ‘up to speed.’ At the other end of the spectrum, you retain creative control but draw a few (or many) of your lectures from this book. Finally, for educated laypersons, the book provides a brief roadmap of computability.”

I have no idea why I hadn't heard of this book, originally published in 1988 and updated with a second edition in 1993, until now. “The New Turing Omnibus” is probably the single closest published equivalent to what I do on this very blog. It's a grab-bag of computing topics. Each chapter is the equivalent of a short blog post examining a particular topic, peppered with tables, diagrams, and illustrations. And the topics aren't presented in any particular order. Browse and find something you like; discard the rest. Here's a short excerpt from Chapter 33, Shannon's Theory—The Elusive Codes:

| Received | Means |
|------------------|-------|
| 000 | 0 |
| 100, 010, or 001 | 0 |
| 011, 101, or 110 | 1 |
| 111 | 1 |

success. For example, if $p = .1$, then the probability that the demon will corrupt the message irretrievably is .028.

The foregoing coding scheme used simple redundancy as a guard against errors. Other, more sophisticated methods are available (see Chapter 12). For example, one may group the message bits into pairs and transmit the pairs along with 2 extra check bits according to the following scheme.

| Message Bits | Check Bits |
|--------------|------------|
| $a_1 a_2$ | $a_3 a_4$ |
| 0 0 | 0 0 |
| 0 1 | 1 1 |
| 1 0 | 0 1 |
| 1 1 | 1 0 |

The first check bit, a_3 , simply echoes the second message bit a_2 ; $a_3 = a_2$. The second check bit is called a *check sum*. It is the logical sum of the first 2 bits; $a_4 = a_1 \oplus a_2$.



Figure 49.1 The noise demon at work

There is no guarantee for any coding scheme that the corruption demon will fail. If it succeeds in changing enough bits, no decoding scheme will be able to recover the original message. For this reason, the message is decoded by the maximum-likelihood method. For each received string of 4 bits, what is the most likely interpretation of the first 2 bits? The following algorithm takes the 4 received bits b_1, b_2, b_3, b_4 as input and outputs b_1 and b_2 , altered according to probable errors detected by the algorithm:

1. **input** b_1, b_2, b_3, b_4
2. **if** $b_4 \neq b_1 \oplus b_2$
 - then if** $b_3 \neq b_2$
 - then** $b_2 \leftarrow \bar{b}_2$
 - else** $b_1 \leftarrow \bar{b}_1$
3. **output** b_1, b_2

The operation of the decoding algorithm is illustrated in the hypercube diagram in Figure 49.2. Each vertex represents a possible 4-bit string to be received. Four of the vertices are circled. These represent the four original (probably uncorrupted) code words.

Vertices representing strings that differ in only 1 bit are connected by an edge. If one analyzes the action of the algorithm on each of the 16 possible received words, one discovers that each word is reinterpreted as the “nearest” code word. The concept of distance applied here is the Hamming distance, the minimum number of edges traversed in going from one vertex (possible word) to a code word.

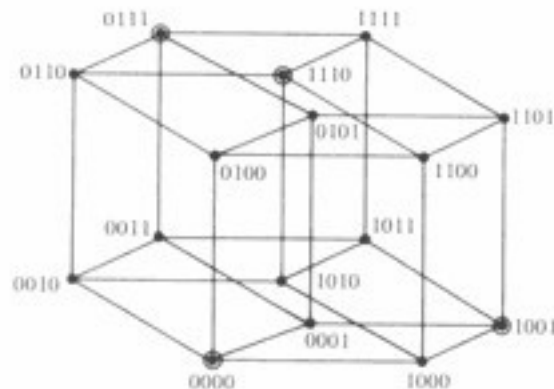


Figure 49.2 An error-correcting code in a hypercube

A [complete table of contents](#) for all 66 chapters of “The New Turing Omnibus” is enumerated at Everything2. I think there's a very high probability that if you enjoy reading this blog on a regular basis, you'll also enjoy this remarkable little book. As promised, it's a great way to [keep practicing the fundamentals](#) for professionals:

“Bert Bates (my co-author) is a blackbelt-level go player, one of the best amateur players in the state. But when a visiting expert—four belt levels above Bert—showed up at the local go tournament, Bert was surprised to see the guy reading a book on fundamental go problems that Bert had read much earlier in his learning. The expert said, ‘I must have read this at least a hundred times. My goal each time is to see how much more quickly I can solve all the problems in the book than I did the last time.’

Some of the best athletes never forget the fundamentals—whether it's Tiger Woods practicing the basics, or a pro basketball player working on free throws. A good musician might still practice arpeggios. A programmer might... I don't know, actually. What would be the fundamentals that a good programmer might forget? I'll have to think about that one.”

But it's not just a book for programmers; it's also got a broad, down-to-earth appeal. It's an intriguing collection of thought puzzles for laypeople with at least a passing interest in the field of computer science.

If you'd like to see more, you can [browse through a few pages of the book at Amazon](#). A few more pages are available [in Google books](#), but beware the randomly inserted "copyrighted image" placeholder instead of the many illustrations and diagrams throughout the book.

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

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

OceanofPDF.com

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.

OceanofPDF.com

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:   

OceanofPDF.com

Other Awesome Books

Hyperink Benefits

- *Interesting Insights**
- *The Best Commentary**
- *Shocking Trivia**

[Use code INKYBUCKS3 to save 25% off your next purchase. Click here and visit Hyperink.com!](#)

[OceanofPDF.com](#)

Copyright © 2013-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!

OceanofPDF.com